

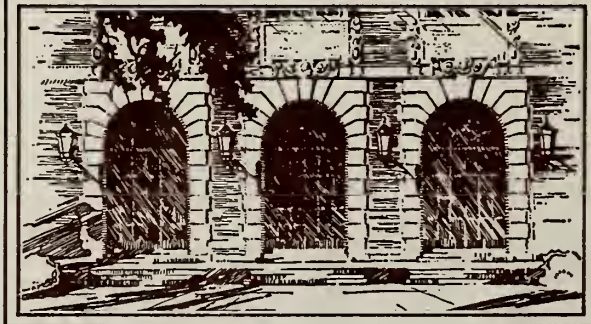
LIBRARY OF THE
UNIVERSITY OF ILLINOIS
AT URBANA-CHAMPAIGN

510.84

IL6r

no. 517-522

cop. 2





Digitized by the Internet Archive
in 2013

<http://archive.org/details/designimplementa520oxle>

Ill
no. 520
cop. 2

UIUCDCS-R-72-520

THE DESIGN AND IMPLEMENTATION OF AN EDUCATIONAL
TIMESHARING SYSTEM FOR THE PDP-11

by

Donald Wayne Oxley

June, 1972

THE LIBRARY OF THE

JUL 5 1972

UNIVERSITY OF ILLINOIS
AT URBANA-CHAMPAIGN



DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

The person charging this material is responsible for its return on or before the **Latest Date** stamped below.

Theft, mutilation, and underlining of books are reasons for disciplinary action and may result in dismissal from the University.

UNIVERSITY OF ILLINOIS LIBRARY AT URBANA-CHAMPAIGN

SEP 1 1972
AUG 29 RECD

UIUCDCS-R-72-520

THE DESIGN AND IMPLEMENTATION OF AN EDUCATIONAL
TIMESHARING SYSTEM FOR THE PDP-11*

by

Donald Wayne Oxley

June, 1972

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

*This work was supported in part by the National Science Foundation under Grant No. US NSF GJ-812 and was submitted in partial fulfillment for the Master of Science degree in Computer Science, 1972.

ACKNOWLEDGEMENT

The author would like to thank Professor Donald B. Gillies for his advice and support throughout the development of this project and the writing of this thesis. Special thanks also go to Dr. Thomas Chen who first realized the feasibility of this project and worked with the author during the early months of development. Al Davis, Stan Kerr, Jim Hart and Keith Phillips have invested an amount of time and effort toward the development of instructional software for the system that cannot be measured. Without them, ETS would have been a total failure; I am deeply indebted to them. Also, the author would like to thank Don Burn, Chuck Collins, Bob Feretich, Barry Finkel, Jim Lellman, David Pasta, Ian Stocks, and the students of Al Davis' CS 201 quiz sections for their help in developing and testing the software. Additional thanks go to Harold Lopeman, Bob Miller, Jim Miller, and Gordy Peterson for their efforts in building and maintaining the hardware.

I am very grateful to Mrs. June Wingler who has done an excellent job of typing both this thesis and the documentation for this project. And most especially, I am indebted to my wife, Pat, for her patience, understanding, and encouragement.

TABLE OF CONTENTS

	Page
I. INTRODUCTION.....	1
II. THE ETS MONITOR.....	5
A. System Resources.....	5
B. System Control Structures.....	7
C. System Modules.....	11
D. System Synchronization.....	20
III. THE ETS FILING SYSTEM.....	33
A. File Organization.....	34
B. File Processor.....	45
C. File Requests.....	48
D. Errors and Protection.....	51
IV. I/O SUBSYSTEM.....	54
A. Basic Principles.....	54
B. Data Structures and Control Routines.....	56
C. I/O Characteristics of Individual Devices.....	60
D. Error Recovery.....	64
V. COMMAND STRING INTERPRETER.....	66
VI. PERFORMANCE AND IMPROVEMENTS.....	71
A. Performance.....	71
B. Tuning the System.....	76
C. Improvements to the System.....	78

LIST OF FIGURES

Figure		Page
II.1	ETS Job Control Structure.....	8
II.2	Job Data Block.....	9
II.3	Basic Software Processors.....	12
II.4	ETS Swapping Processor.....	14
II.5	XRB Structure.....	17
II.6	Physical Character Oriented Device I/O Control.....	19
II.7	System Sequencing.....	27
III.1	Master File Directory Structure.....	35
III.2	User File Directory Structure.....	39
III.3	File Control Block.....	43
III.4	UFD Window Turn.....	44
III.5	File Processor Flow Diagram.....	46
III.6	FIP Control Information and Utility Routines.....	47
III.7	FIRQB Structure.....	50
III.8	File Protection Codes.....	52
III.9	FIP Error Calls.....	53
IV.1	Line Printer Device Data Block.....	57
IV.2	Device Priority Assignments.....	61
V.1	System Commands.....	70
VI.1	Execution Time for Some System Routines.....	72

I. INTRODUCTION

ETS is a general purpose timesharing which has been implemented with educational goals in mind. The basic design of ETS is taken from the monitor of the Resource Timesharing System, RSTS-11, written by Nathan Teichholtz of Digital Equipment Corporation.

The word 'educational' is used in the name of the monitor for two purposes:

ETS assumes that its users may be students who have never seen a computer. Consequently, the necessary operations performed by the students are kept to a minimum (e.g., it requires at most three standard commands to connect the student with the system and begin execution of one of the user packages). The system is also careful to assume that any input the student may type may be garbage and if so, it must be ignored. Unfortunately, ETS as yet does not make an attempt to understand a student's incorrect inputs.

ETS is in principle a monitor which a moderately experienced student can read and understand. A great deal of this author's effort has gone into internal documentation of the code in an attempt to make it as easy to follow as possible. ETS currently comprises about 9000 lines of code (including the documentation which is about 37% of the lines) which is quite a chore for even the dedicated student to understand. This author feels that it is not necessary to study the entire monitor if a student is only interested in a certain feature; however, a student who wishes to make

any extensive changes should probably have a basic idea of all of it. Ideally, the maintenance of ETS is done by students who are studying operating systems and need some actual experience. Since undisciplined changes to the basic monitor cannot be allowed without some supervision, it is recommended that ETS' maintenance be handled in a manner similar to that of many other operating systems:

Students should plan the changes they want to make and write them up both in terms of what they want to do and how they want to do it. After the proposal is criticized by a separate group of students, the changes may be made to a new version of the monitor and the necessary changes to the documentation should be prepared. Again a second group of students is called in to test out the modifications. If they certify the changes correct, the new system becomes a provisional system and the documentation should be updated. After a fixed period of time, the provisional system becomes the standard system and new modifications may be started.

ETS is not now and probably never will be a completed system. Part of its design philosophy is that it is to be an evolving system which accepts new changes as they come along. Many of the algorithms in the monitor are not the most sophisticated known (no disk optimization is done, scheduling of jobs is round robin). It would be particularly instructive for students to code some of the more powerful algorithms and compare their performance with the simple algorithms used in ETS. This author suspects that the simple schemes employed in ETS will fare quite well; not because they are necessarily better, but because the ETS environment is such that there are not enough different conditions around at any one time to use more powerful schemes efficiently.

There are, however, many things which the monitor does need (e.g., better buffer accounting, dynamic swap area allocation) which would make good projects for students taking classes in operating systems.

Students trying to understand ETS will need both this thesis and, more important, a copy of the code. It is recommended that a student read this thesis once and then reread it while locating in the code some of the routines mentioned. The processing of a specific input should be followed until it is completed; then a different input should be followed. It should be noted that there are two main 'drivers' to the system--clock interrupts and EMT's. When a student can follow all branches generated by these two interrupts, he will have a basic understanding of ETS. In order to improve the documentation of ETS, students should note their problems in understanding the system, and as they solve them, document them so that other students need not fall in the same traps.

ETS assumes two types of users:

Students who are using the facilities provided by the system. These are documented in the "Student User's Guide to ETS." When servicing this type of user, ETS is providing a service and can run without intervention on the part of systems programmers.

Systems programmers who are either writing new facilities for ETS or are working on the system itself. Coding techniques for systems programmers are documented in the "System Programmers Guide to ETS." Because of the lack of memory protection on the PDP-11, much of this type of work must be done when other users are not present.

This manual divides ETS into four major parts:

The timesharing monitor discusses the routines which actually enable timesharing to be carried out.

The filing system explains the organization and usage of the disk.

The I/O subsystem describes the handling of devices other than the disk.

The command string interpreter discusses the control facilities given to the user over his job.

II. THE ETS MONITOR

The monitor in any timesharing system has a number of basic characteristics. It is the heart of a software system which allows multiple users to utilize the computer without considering the relationship of their programs to that of other users. A timesharing system is much like a multi-programming system except that it has the additional responsibility of providing rapid user response in an interactive fashion.

All timesharing systems contain routines to choose the next user to be permitted to use the CPU, routines to allocate core memory (if all jobs are not resident simultaneously) and handle swapping between core and secondary memory, routines to handle physical I/O, and routines to handle the filing structure. When considered individually each of these routines is a more or less simple piece of code to do a particular, well-specified job. It is not the structure of the monitor routines themselves which makes a timesharing system--it is their interconnection which is important.

In describing the monitor we shall first introduce the available monitor "resources," then the monitor control information, then a brief description of the individual monitor parts, and finally the interrelationships between the parts.

A. System Resources

It is the responsibility of the operating system to supervise the allocation and usage of the system resources. There are five basic resources

which the system looks after:

1. CPU Time

The system must see that each task receives a reasonable amount of CPU time in a given interval of real time. ETS generally tries to give each user a 100 msec timeslice in round-robin fashion.

2. Core Memory

Since all or part of a job must be in directly accessible storage for execution, the system must allocate core to a job in order for it to be run. ETS will allow a job to have any amount of core it wishes (up to a preset maximum), in increments of 4000_8 bytes.

3. I/O Device

With the exception of the disk and DECTape, all ETS I/O devices may be accessed by only one user at a time. A user wishing to use an I/O device requests that it be assigned to him. If it is available, ETS will assign it to him; otherwise, the user must wait until it is free.

4. Buffers

Since users may be swapped out during times when I/O is taking place, or the system is providing service, it is necessary to have core resident data areas. System buffers provide this service. ETS maintains a pool of ~100 buffers which users may request when needed.

5. Disk

ETS uses the disk for two purposes:

a. Back-up Memory

When a user is not in core, his program is kept in a portion of disk called the swap area from which it may be retrieved for his next timeslice.

b. User File Storage

Active users may read their program files onto disk while operating on them. Handling of file storage is very important and is treated as a separate section of this manual.

B. System Control Structures

In order to function properly, a system must keep available a wealth of information on what is going on in the system at any given time. This information is a common pool of data which system routines may access and must update with any information they have which other routines may need to interrogate.

1. Job Data Blocks

Associated uniquely with each job (user) logged in on the system are two 16 word data blocks. The first (called the Job Data Block) contains a pointer to the second (called the I/O block or IOB) as well as information relating to the job's status. The second block contains a list of "channels" (words which may contain pointers to I/O control blocks). There is a pointer entered in the I/O block for each file or device which the user has open. (See figures II.1 and II.2.)

2. File Control Blocks (FCB)/Device Data Blocks (DDB)

Associated with each open file and each device in the system is a 16 word control block. When a user opens a file (device), a pointer to the FCB/DDB is stored in the appropriate channel of his I/O block. A FCB/DDB gives status and access information for its particular file (device). Note that there is no FCB for an unopened file, however, the system still maintains a DDB for each device whether it is open or not.

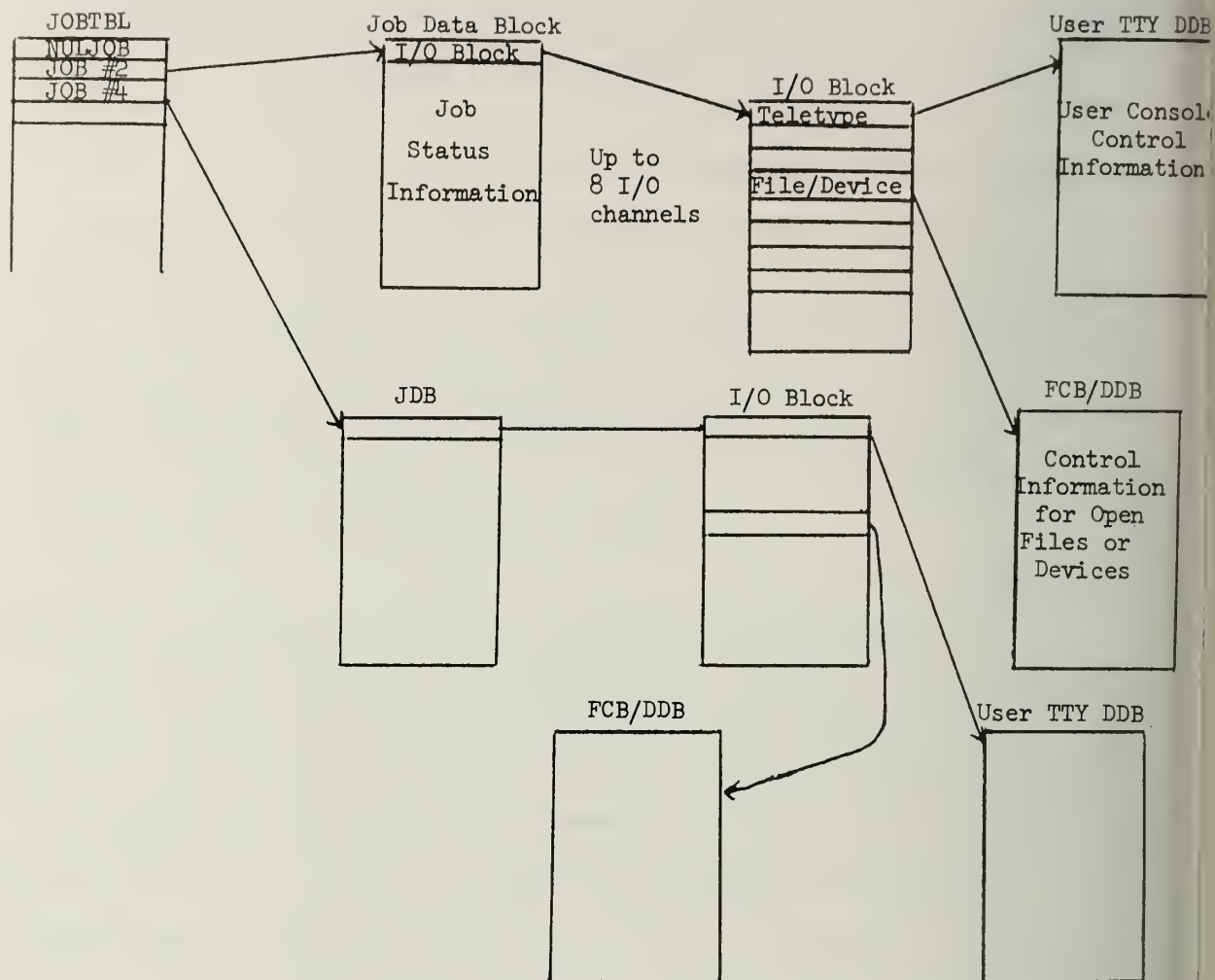


Figure II.1 ETS Job Control Structure

JDIOB	-	Pointer to I/O Block	0
JDFLG	-	Job Status Flags	2
JDIOS	-	I/O Error Status	4
JDSP	-	Relative User Stack	6
JDCPU	-	CPU Time Used	10
JDKCT	-	KILO-CORE-Ticks Used	12
JDDEV	-	Device Time Used	14
JDSIZ1	-	JDSIZ0 Next Size in K Present Size in K	16
JDUFD	-	Start Block of Job's UFD	20
JDPPN	-	Proj-Prog Number	22
JDSGNW	-	Unused	24
JDSGMX	-	Unused	26
JDUFLG	-	User Communication Flags	30
JDNAME	-	Job Type Code	32
		Unused	34
		Unused	36

Figure II.2 Job Data Block

3. JOBTBL

At initialization time, the system establishes a table of (possibly null) pointers which will be used to point to the first Job Data Block (JDB) for each active job. There is a pointer position for each possible job in the system. When a job is created, and its Job Data Blocks are established, the appropriate entry in JOBTBL is loaded with a pointer to the JDB. Thus at any time, we may interrogate JOBTBL to find the number of active jobs in the system, as well as pointers to their control blocks. (See figure II.1.)

4. DEVTBL

During initialization, a DDB is set up for each system device. Pointers to these devices are kept in DEVTBL. There is a unique entry for each device. Through DEVTBL, device service routines can record transaction results without knowing the owner job. When a device is opened by a user, note that a separate pointer to the DDB is placed in the user's IOB so that device independent routines can communicate with an arbitrary device without worrying about what it is.

5. JBSTAT-JBWAIT

As it is quite possible for a job to request input from a currently inactive device or output to a busy device, it is necessary to have a mechanism for causing jobs to wait on the completion of an I/O request.

This mechanism is implemented by assigning to each possible blockage condition a bit position from 0-15. We then associate two status words with each job.

In the first--JBWAIT--we set a bit indicating any blockage condition that arises. When the condition no longer exists, the controlling service routine sets the same bit in JBSTAT.

We then define the condition that a job be runnable as

$$JBSTAT(JOB\#) \wedge JBWAIT(JOB\#) \neq 0.$$

6. CORTBL

Independent of job status information, it is necessary for the system to keep information on each available core segment. CORTBL is a table with one entry of two bytes for each 4000_8 bytes of user core. The low byte of each entry contains the job therein resident. The high byte contains status information on that block of core.

7. Volatile System Information

The system needs a number of variables which define its "state" at any given instant in time. These include such things as time and date, the current job with pointers to some of its relevant information, headers for system queues, file processor status, and swap processor status.

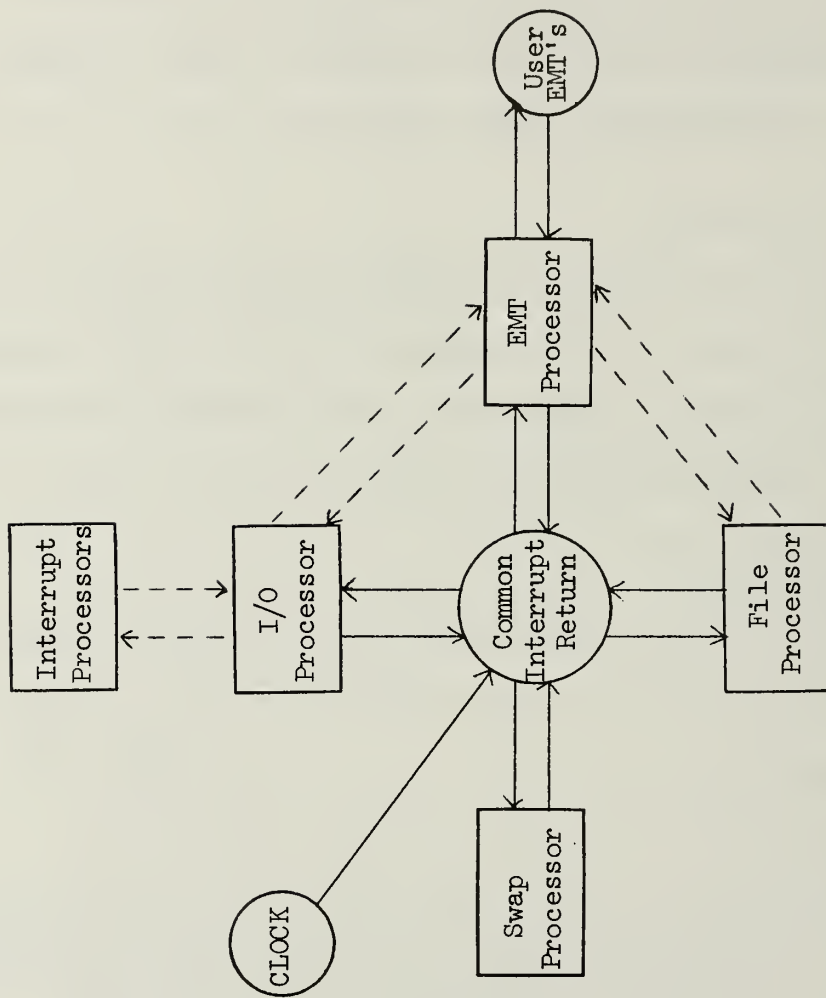
8. File System Information

The system also has a fairly complex data structure to locate and control items in its filing system. The discussions of such structures will be left to chapter III.

C. System Modules

ETS uses five basic software processors to accomplish timesharing. (See figure II.3.) In general, each processor operates asynchronously from the other processors. Some sequencing and interlock mechanisms do exist and are explained in section D.

We shall examine each processor, outline its basic functions and how it accomplishes them.



↑ = Direct Transfer of Control
 ↑ = Communication of Control
 - - - with Flags

Figure II.3 Basic Software Processors

1. Swapping Processor

The swapping processor is responsible for the orderly distribution of CPU time among available jobs. As a consequence of this, it must also allocate/deallocate core memory.

Figure II.4 is a simplified flow chart of the swapping processor. When the processor is initiated, control passes to a routine (SAVJOB) which will stop the current job. Next control passes to the scheduler (SCHED) who selects a job to be run. If the selected job is core-resident, then control passes to RESJOB which will initiate the new job. If the selected job is not core-resident, SCHED calls SWAP to bring the job into core. If there is an empty place in core large enough for the selected job, SWAP will bring it in; if not, SWAP will repeatedly call MAKERM to force a user out until he has enough space to bring in the selected job.

Once a job has been located and made core-resident, it will be restarted and given a timeslice of 100 msec.

2. EMT Processor

The Emulator Trap software interrupt provides the basic means of communication between the user and the system. User I/O, file, and service requests are handled by the EMT processor.

The EMT processor will save a job's registers, determine the particular function to be performed and will transfer control to the service routine for the function in question.

If the EMT call is for the file or I/O processor, control will be passed on to it; otherwise, the particular service routine is involved directly.

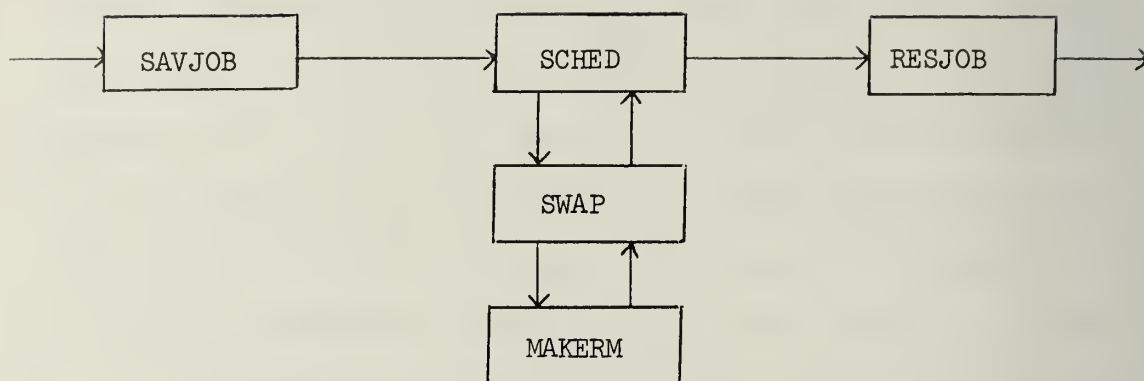


Figure II.4 ETS Swapping Processor

Presently, there are the following EMT routines:

- a. .BREAK - to allow a job to be stopped temporarily on command, control will be returned to the scheduler. The job will be restarted the next time the scheduler encounters it in the queue.
- b. .CORE - a job may request that he be assigned n blocks of core from his present size m . For $m > n$, the last $m - n$ blocks of core are removed from the end of the job. For $n > m$, the job is swapped out and brought in at the next size with the blocks added to the end of the job.
- c. .EXIT - programmed termination.
- d. .FIND - request to position a file at a particular block.
(See chapter III.)
- e. .LOCK - request to set lock bit on job's entries in CORTBL.
- f. .TTAPE - request to change teletype to tape mode of input.
- g. .TTRST - request to change teletype to normal operation.
- h. .ULOCK - request to release core a user may have locked.
- i. .WAIT - request to wait on the appearance of a delimiter in the teletype buffer.

3. File Processor

See chapter III, the ETS Filing System.

4. I/O Processor

The ETS I/O processor is comprised of a standardized "front-end" which communicates with the user and of separate device service routines for each type of device in the system.

Within reasonable limits, ETS I/O structure is device independent. A user simply constructs a transfer block (XRB) containing the necessary I/O parameters and executes a .READ/.WRITE passing the address of his XRB to the system.

a. I/O Calls

The user must allocate in his own area a transfer block (see Figure II.5) and insert the appropriate parameters. He then places the address relative to the beginning of his area of the XRB in R3 and executes a .READ or .WRITE.

b. Parameter Decoding

ETS has a controlling I/O dispatch program (USERIO) which will check the I/O parameters supplied by the user for errors (does he want to write on the card reader?). If the user has asked for a valid operation and has the necessary files/devices open, USERIO will select the appropriate device service and pass control to it.

c. Device Service Routines

Associated with each device is a unique service routine which can cope with the special needs of the individual device.

The service routine will attempt to transfer characters to/from the monitor chain buffers (see section IV.B.2) from/to the user area. If the chain buffers become full/empty, the device service routine will fire up the device interrupt service to empty/fill the buffers and will put the job to sleep to wait on the physical I/O completion.

In order to provide a fast response to the user, I/O is considered to be a process between the user and the system. The system then uses its own interrupt service to do the actual I/O transfers. This technique allows users at TTY consoles to continue typing in

XRB:	XRLNK (=0)	-	System Link Word
	XRPTR (=2)	-	Offset from XRB to BUFFER--not Modified
	XRLLEN (=4)	-	Maximum Transfer Length for Read
	XRBC (=6)	-	Actual Transfer Count
	XRLOC (=10)	-	Offset from XRB to BUFFER--Updated
	XRCI (=12)	-	Channel for Operation
	XRPPAR (=14)	-	1st Parameter Word
	XRPAR2 (=16)	-	2nd Parameter Word

On read operations, XRBC should be initially zero and the system will return the actual character transfer count. In the case of line oriented devices (e.g., TTY) a read operation will terminate on the transfer of a <CR> unless XRLLEN is exceeded; otherwise, the system will fulfill XRLLEN unless an EOF is encountered. In the case of character (non-disk) I/O, XRLOC will point to the last character transferred.

Figure II.5 XRB Structure

their next request while the present request is being processed. The system will simply store the request in its own buffers and pass it on to the user program when requested.

In order to prevent unnecessary "sleeping" of jobs, this same technique is used for output. When a user program wishes to perform output, this output is simply transferred to the system buffers and the job is allowed to continue processing. In order to prevent the user from outputting more than the system can handle, each device has associated with it a maximum number of buffers which may be used at any one time. If a user job needs more than this, it is forced to sleep until the system can catch up with some physical output.

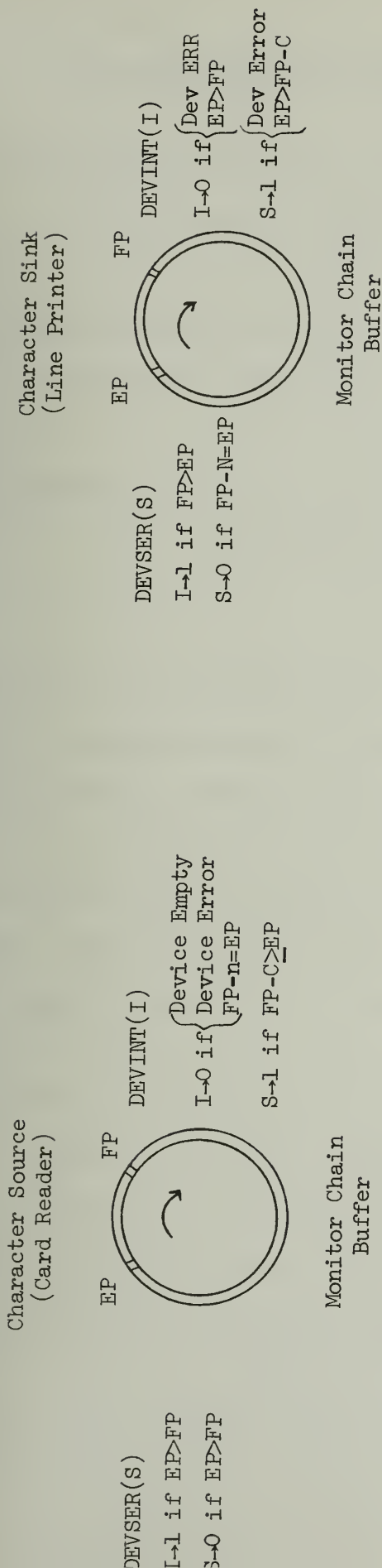
5. Interrupt Processors

ETS communicates with the outside world through the interrupt mechanism. All I/O devices are run in interrupt mode and communicate with their individual handlers which are responsible for interfacing with the system.

Interrupt drivers are triggered by device service routines and will run until they deplete a necessary resource (e.g., buffer space) at which time they will cause the initiating job to be rescheduled. Communication between the device interrupt handler and the device service routine is diagrammed in figure II.6 utilizing the monitor chain buffers.

The asynchronous operation of the device interrupt handlers and the device service routines is achieved by using a chain of system buffers (Monitor Chain Buffers) to store data on a temporary basis. These buffers are manipulated with two routines:

STORE - will store a character into the next available
position in the chain buffers. If a particular



EP=Empty Pointer
 FP=Fill Pointer
 N=Max Length of Circular Buffer
 C=Number of Characters to
 Enable DEVSER and DEVINT to
 Run Simultaneously

S=DEVSER Switch - S=0 ⇒ Unable to Run
 I=DEVINT Switch - I=1 ⇒ Running
 DEVSER=Interface Routine between User and
 Interrupt Handler
 DEVINT=Device Interrupt Service

Figure II.6 Physical Character Oriented Device I/O Control

buffer is full, it will allocate another up to a preset maximum.

FETCH - retrieves characters from a chain buffer until no more remain, thus causing a job or device to be restarted.

All buffers are allocated from the system buffer pool and the pointers for a particular chain buffer are stored in the DDB of the device in question.

The interrupt handlers are also responsible for checking the device status and providing error recovery facilities.

D. System Synchronization

Any system which attempts to handle multiple tasks concurrently must insure that one task cannot interfere with the execution of another. More important, no task should even be aware that there is another task in existence competing with it for resources.

The system, of course, must be aware of all tasks and must provide mechanisms for stopping and restarting jobs and switching tasks. Since redundant service facilities (file handling, I/O service, etc.) are impractical in all cases, impossible in many, the system must be especially careful to see that these facilities can handle multiple users. This is done by: (1) providing a queue of work for a particular facility so that it may take its tasks one at a time, (2) assigning the facility to a particular task (e.g., card reader and service routine) so that it services only one user, and (3) making the facility reentrant so that users may execute it concurrently.

ETS uses a number of different mechanisms to synchronize itself:

1. Module Design Independence

This mechanism is simply a matter of design philosophy. Modules within the system are written with carefully specified and strictly adhered to communication protocol. In a system as cramped for space as is ETS, there is considerable tendency to save code by ignoring protocol and ETS is certainly guilty of this at times; however, on the whole, ETS routines do not make unspecified and uncontrolled assumptions about other routines.

In order to cut down as much as possible on unnecessary routine interconnections, ETS tries to follow these design criteria:

- a. In order to cut down on critical segments (code segments which may not be interrupted either because the code is self modifying or it modifies multiple state variables which have to maintain some specified relationship), system routines are reentrant where possible.
- b. As few routines as possible may modify any given system variable.
- c. Any routine containing a critical segment is responsible for protecting that segment, i.e., the calling routine does not have to change priority.

2. Queueing

When a number of similar functions are scheduled, ETS uses a queue in order to decide which function to execute next. There are three types of queues used by ETS:

a. FIFO Queue

This type of queue is used by routines requesting a service from some support facility (e.g., a disk transfer) and is the standard queue used in ETS.

b. Round Robin

The system uses a round robin queue to scan fixed lists of tasks to determine which need service. JOBTBL which is a list of jobs in the system is scanned in round robin fashion when the system wants to find the next user job to be given the CPU. Scanning of the round robin queue always picks up from the place where the previous scan left off in order to divide system resources equally.

c. Modified Round Robin - L3QUE

The system keeps track of the major tasks ahead of it with a modified round robin scan. Each of the tasks (e.g., scheduler, swapper, file handler, etc.) is assigned a bit in the queue word L3QUE. When it is necessary to schedule service from one of the major tasks, the appropriate bit is set in L3QUE.

Every time control returns to the routine RTI3 at a priority ≤ 2 , L3QUE is scanned from right to left. If any non-zero bit is encountered, it is cleared and control is passed to the appropriate routine, which will eventually return to RTI3 permitting L3QUE to be scanned again.

3. Device Assignment--Protection Codes

Since most devices and many files are capable of being used by only one job at a time, ETS sets a job code in the DDB of devices and the name block of files which indicates that a device/file is or is not available to a particular user. Before using a device/file, a user must successfully open it (see section III.A.2). If the device is already in use (another job has already opened it), the attempted open will return an error. In the case of files, it is possible for more than one user to read a file, but only one may be write enabled on a file. A file may also be made unavailable to

other users by setting the appropriate protection codes. If a user attempts to open a file on which he is not read or write enabled and he is not the owner, the open will generate an error.

4. Job Blockage, Relocation, Memory Protection

In order to implement timesharing, it is not sufficient to be able to start, stop, and switch jobs. Jobs will often generate conditions which require them to wait on some later event (e.g., completion of a line of TTY input). It is very inefficient to simply keep restarting the job to allow it to check for the occurrence of the given event; a better solution is to set some type of flag and let the system watch for the event and then wake the job when the event has occurred. In ETS, there are a number of specific events on which a job might want to wait. (In particular, each device has a unique event associated with it as does the file processor and the timer service.)

When a job is created, it is assigned two status words (JBSTAT and JBWAIT) to indicate blockages that may occur. When a job is running, both JBSTAT=1 and JBWAIT=1. When a job requests a service for which it must be blocked, JBWAIT is cleared and the bit associated with that event is set. When the event occurs, the same bit is set in JBSTAT. The condition that a job be able to run is then $JBSTAT \sim JBWAIT \neq 0$. The system can then check $JBSTAT \sim JBWAIT$ and schedule a job to run only when it is not blocked. Note that a job will then run when any one of a number of blockages are relieved, not when all are relieved. Note also that there are separate JBSTAT and JBWAIT words for each job.

When a job is blocked, ETS will most likely swap it to disk to await the relief of its block. When the job becomes able to run again,

ETS must bring it back into core and would like to put it in any available space rather than try to put it back exactly where it was. Since the PDP-11 has no hardware relocation, some scheme must be used to allow the job to keep all of its addresses straight. Even though relative addressing is simple on the PDP-11, it is not feasible to tell the user that he cannot use absolute addresses, especially since the system hardware pushes absolute addresses on the stack.

ETS uses a scheme which allows the user a reasonable freedom in developing addresses, but the user must be aware of the needs of the system and vice versa. ETS provides the user with three areas where he may develop absolute pointers:

a. The User Stack

When ETS moves a job in core, it will conditionally relocate everything in the user area from the top of the stack to the stack base by an amount equal to the distance a job is moved. Since a job is always started at a 4000_8 byte boundary, all changes will be in the upper 5 bits of the word. Before relocating a given word, ETS will check that it is a possible address somewhere in the user area (presently 30000_8 - 60000_8). If it could not be a pointer into the user area, it will not be relocated. This relocation restricts the user from placing any datum on the stack which may be interpreted as a pointer (i.e., $30000_8 \leq \text{datum} < 60000_8$). Note that the PS which the system hardware pushes on the stack will never be interpreted as a pointer.

b. Seven Relocating Pointers

Since the user may wish to develop a pointer offset which does not point into the user area, the system provides 7 words in system core which are always relocated.

c. Registers

Since it is often desirable to place absolute pointers into registers, the system provides a means by which it conditionally relocates registers if the user has indicated that they contain absolute pointers. Associated with each register R0-R5 is a byte ($\text{REGREL}+i \Leftrightarrow R_i$) in system core. When the system wishes to relocate a user, it will consult the relocation bytes and relocate any register for which $\text{REGREL}+i \neq 0$.

Since it is unreasonable (and suicidal) to expect beginning students to code without errors, and to code under ETS' coding restrictions, all user written code will be run under an interpreter. Once code has been completely debugged and modified to ETS' conventions, it may then be run without an interpreter.

5. Priority Control

The PDP-11 hardware provides 8 levels of job priority. ETS breaks this into 3 operating ranges:

a. Levels 0-3

Normal interruptable code (both system code and the user job) executes in this level. Priority arbitration within these levels is handled by the routine RTI3. All software interrupts, the clock, and the disk may cause control to pass to RTI3.

RTI3 will automatically return control to the interrupted process if it was running at a priority ≥ 3 . Otherwise, it will process L3QUE to see if there are other tasks to be performed.

b. Levels 4-6

Device interrupt service routines run at these levels. Priority arbitration is by the hardware.

c. Level 7

Critical code segments run at priority 7 so that they may not be interrupted.

6. Flow of Control

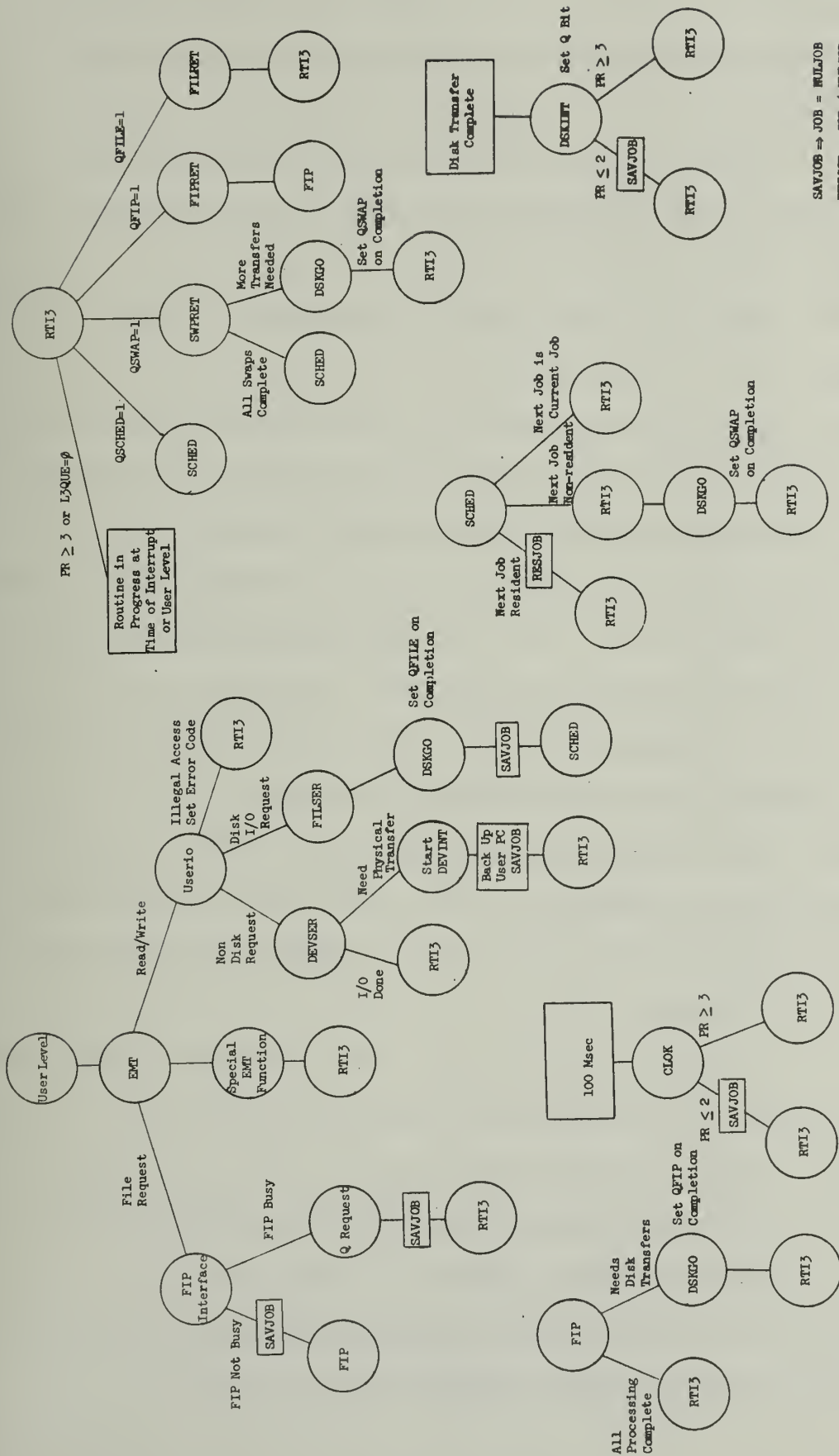
(Note: this section relies heavily on the diagrams in figure II.7 and assumes a reasonable idea of both the purpose and structure of the routines mentioned.)

Flow of control in ETS may be thought of as a tree in which each external leaf on every tree is the root of some other tree (see figure II.7). Each node in the tree is a routine which may call one or more "lower level" routines. At any point where a node has 2 or more daughters, there is some condition or event which may be interrogated to determine which branch to follow.

Notice that there is only one tree with a root of user LEVEL--this node is the user program. At any given time there is only one active user which is represented by this node. Since no other user is active, control cannot possibly pass out of these trees to another user. Any time that the scheduler (SCHED) is permitted to run, it may make another job the active job.

These diagrams also do not show any device interrupts other than disk and clock. No other devices may cause control to be transferred to any part of this tree directly. Their interrupt handlers may simply store or fetch a character and (possibly) set a bit in JBSTAT, after which they must return to the interrupted process.

There are two routines which are "operator-like" in nature. SAVJOB will cause the current job to be the NULJOB (if the current job is



SAVJOB ⇒ JOB = MULJOB
RESJOB ⇒ JOB ≠ MULJOB

Figure II.7 System Sequencing

already NULJOB, it has no effect). RESJOB will cause the current job to be anything other than NULJOB (if there is no job ready to run, it will not be called).

Normally, the system is executing code associated with the user LEVEL (user). This code runs at priority 0 and may be stopped at any time.

There are 3 events which may cause control to pass out of user LEVEL:

- a. A system tick may occur. The clock interrupts every 16 msec. Every 6th interrupt (100 msec) causes a system tick during which the clock will set the QSCHED bit in L3QUE so that the system will choose a new task at the next opportunity. If the priority at the time of the interrupt was ≤ 2 , the clock will execute SAVJOB and then branch to the common interrupt return RTI3. If the CPU priority at the time of the interrupt was ≥ 3 , the clock will branch directly to RTI3 which will return to the interrupted process.
- b. A disk access completion interrupt may occur. The disk will set the Q-bit in L3QUE (QSWAP, QFILE, QFIP) for the service routine which requested the disk transfer, execute SAVJOB if the CPU priority was ≤ 2 , and branch to RTI3.
- c. The user may execute an EMT. Unlike the disk and clock interrupts over which the user has no control, the EMT is a request by the user for some type of system support.

There are 3 types of support which a user may request:

(1) File Processor Call

The user wants a service provided by the file processor (FIP) such as creating or opening a file.

FIP operates in a strictly serial fashion on all user requests; thus, the user's request passes to a special FIP interface which checks to see whether or not FIP is currently busy with another request. If it is not busy, the job is saved and processing is started. Note that FIP generally requires one or more disk accesses to complete its work; consequently, the job is dumped and the system will look for something else to do while FIP is executing. If FIP is busy, the request is queued for FIP to find later, the job is saved and control passes to RTI₃. In this case, the calling priority of RTI₃ will be that of the user job (PR₀), not that of the FIP interface; thus the job will be stopped and the system will look for another job to run.

(2) Special EMT Function

The user wants a service provided by one of the special EMT functions (see II.C.2) such as .LOCK or .WAIT.

These functions are executed at PR₃ and when complete, they return to RTI₃. Note that the calling priority in RTI₃ will again be PR₀ (SVC LEVEL) so the job may be dumped. However, it is likely that the QSCHED bit has not been set and control will soon get back to the same job.

(3) USERIO Call

The user requested an I/O transfer from disk or device. There are 3 possible paths:

- (a) The user has requested an illegal transfer. The user may have attempted to operate on an unopened device or file or he may have attempted an uncool like reading from the line printer. In either case, he will be given an error

code and control will be given to RTI3. The calling priority will again be level 0.

(b) The user has requested a file transfer (I/O to or from disk) control will pass to the file service routine (FILSER) which will set up the parameters for the transfer. Control will then pass to DSKGO which will queue the transfer on the list of things the disk is to do (if this is the first item in the queue, it will be started). The job is saved and control passes to the SCHED to find something else to do.

(c) The user has requested a device transfer from a non-file oriented device. Control will pass to the appropriate device service routine for the device in question. The service routine will attempt to complete the transfer based on the contents of its monitor chain buffers. If it completes the transfer, it returns through RTI3. If it cannot complete the transfer without doing some physical device transfers, it will fire up the device interrupt handler to do the physical transfer. Since it does not want to wait on the device to complete the transfer, it simply backs up the user PC by two (all I/O calls are one word instructions) so that the user will re-execute the call, sets the appropriate blockage bit in JBWAIT, saves the job, and branches to RTI3. Note that serial I/O calls are structured such that at each call to DEVSER, the transfer may be resumed from the point at which it left off previously, thus a single I/O request can cause multiple physical transfers.

The common interrupt return RTI3 is the basic dispatch point for all system functions. RTI3 dispatches control based on the CPU priority at the time of the interrupt which transferred control to it and the contents of L3QUE.

a. CPU Priority ≥ 2 or L3QUE=0

Control returns to the interrupted process. Note that even though interrupts may be nested, since all interrupt service routines run at priority 3 or higher control will return to the previous interrupt service. Note also that SAVJOB (which is never executed by any nested interrupt) switches to NULJOB by making it look like NULJOB was the interrupted process--not the user.

b. CPU Priority ≤ 2 and QSCHED=1 in L3QUE

Control passes to the scheduler which attempts to find a job to run. SCHED has three alternatives:

(1) The next job to be run is the current job. Control is simply returned to RTI3 and the scheduler has no effect.

(2) The next job to be run is not the current job. If the current job is NULJOB, then control passes to RESJOB to establish the new current job. If the current job is not NULJOB, then the flag JFSTOP is set so that the job will be stopped at the next system tick. Control can then return to SCHED and the new job will be established. In either case, control returns through RTI3.

(3) The next job is non-resident. Control will pass from SCHED to the swap processor (SWAPAR) which will attempt to swap the next job into core. SWAPAR will initiate any disk transfer that is necessary (it may be necessary to swap some core resident jobs out first) through DSKGO and then return through RTI3.

c. CPU Priority ≤ 2 and QSWAP=1 in L3QUE

SWPRET is the return point for a previous call to SWAPAR which is queued only by a disk completion interrupt for a transfer initiated by a job swap. SWPRET will check if all swaps asked for by SWAPAR are complete, if so, it will return to scheduler. If not, it will choose the next swap, fire it up, and return through RTI3.

d. CPU Priority ≤ 2 and QFIP=1 in L3QUE

FIPRET is the return point for the file processor. It will check for a successful disk transfer and then return to FIP at the point from which the previous operation left off.

e. CPU Priority ≤ 2 and QFILE=1 in L3QUE

FILRET is the return point for FILSER requests. It will check for a successful transfer, relieve the disk blockage in JBSTAT for the appropriate job, and return through RTI3.

III. THE ETS FILING SYSTEM

The ETS filing system is a complete management package for the disk file storage. Because of severe space limitations, it does not encompass DECTape or other devices as yet.

All files in ETS consist of logically sequential blocks of 256 words each, numbered in order beginning with zero. Through the use of directories containing retrieval links and a mapping scheme from logical to physical blocks, ETS may dynamically allocate/deallocate blocks for any file without causing garbage collection problems.

The ETS file processor (FIP) is a separate subsystem from the monitor which operates asynchronously from the monitor and services all users one at a time on a first come-first served basis. FIP maintains its own stack and switches to it whenever it is busy and switches back to the current stack when it either completes processing or has to wait on the disk.

Only a small part of the FIP code is core-resident. The majority of it is disk resident and is brought into a special buffer (FIPBUF) when it is to be used. The code buffer is also 256 words long so each block of non-resident code fits nicely into a disk block. Similarly, FIP maintains a 256 data buffer (FIBUF) which it uses to store disk blocks which it needs to access on a temporary basis (e.g., a user directory).

A. File Organization

This section will discuss the data structure used to control the filing system.

1. Directory Structure

In order to facilitate protection and isolation of different user groups, ETS classifies users according to a "project" and according to "programmer" within a project. Thus each user may be assigned his own "project-programmer number" (PPN) which is a unique identifier of a special directory containing his files. Since file names need to be unique only within a particular PPN, this allows programmers to name files without worrying whether that name is used by some other programmer.

Associated with each PPN is a unique directory called a User File Directory (UFD) which contains information on files belonging to a particular user. The system then maintains a Master File Directory (MFD) which lists all UFD's. Each file has associated with it a number of 8-word blocks which contain all the information necessary for describing and finding any part of that file. The first of these is a name block which (obviously) contains the file name as well as pointers to other information. The second is the accounting block whose use is again obvious. The third is a retrieval block which will be explained later.

a. Master File Directory

The Master File Directory is structured much like the User File Directory; however, instead of containing pointers to individual files, it contains pointers and information about the User File Directory associated with each project programmer number (PPN) known to the system. The first entry in the Master File Directory is the name block for PPN [1,1] (see Figure III.1). The first entry in this name

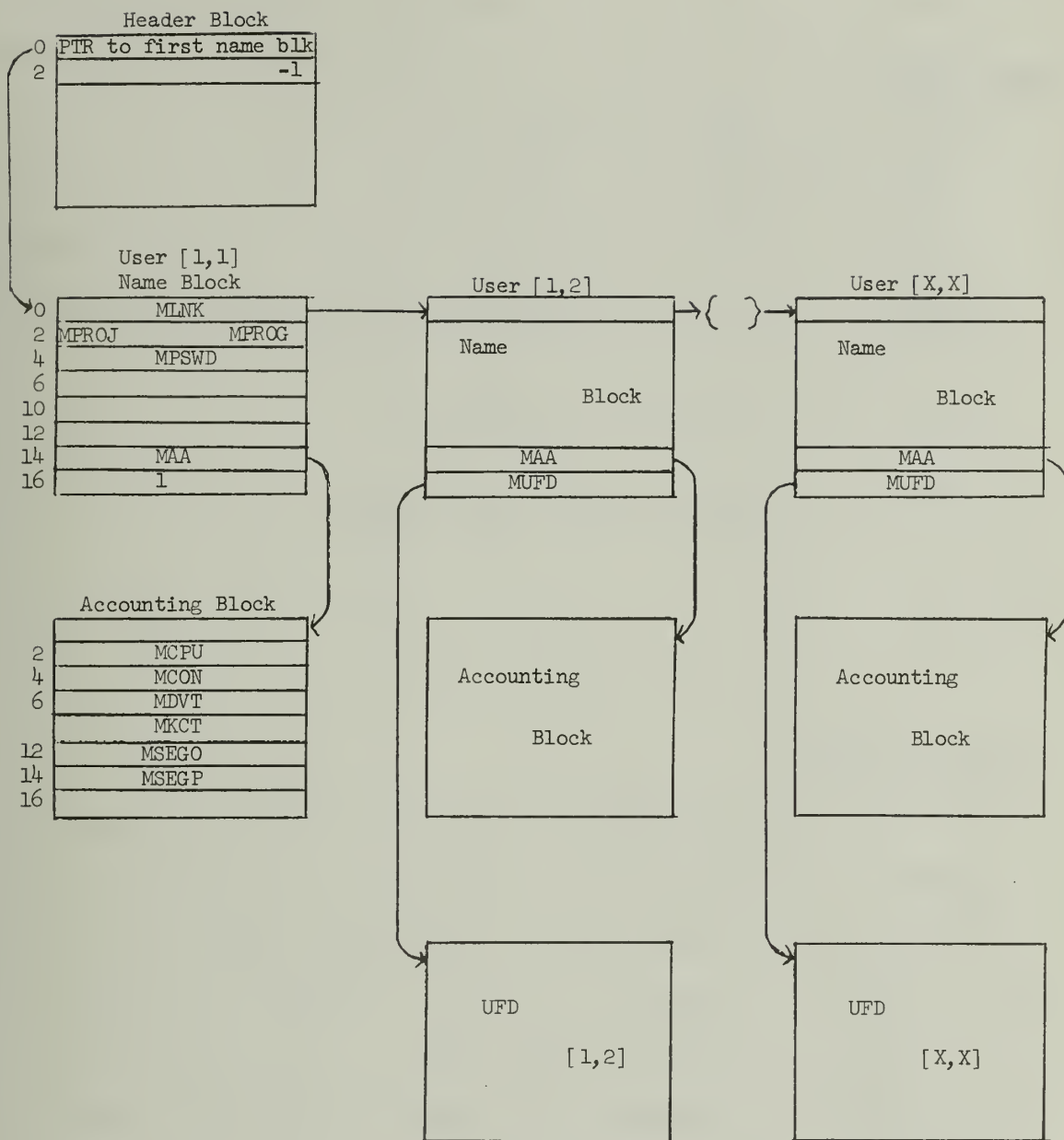


Figure III.1 Master File Directory Structure

Name Block

MLNK=0	;Link to next name block
MPROG=2	;8-bit programmer group #
MPROJ=3	;8-bit project number (a group is a suborder of project)
MPSWD=4	;6-character password which is used when logging in under a specific Project-Programmer #
MAA=14	;Pointer to Accounting Block (Actual Logic-Address)
MUFD=16	;Pointer to UFD header block (Physical Disk Segment Number)

Accounting Block

MCPU=2	;Accumulated CPU time under account
MCON=4	;Accumulated connect time
MDVT=6	;Accumulated device (peripheral) time
MKCT=10	;Accumulated KCT's (KCT = 100 msec)
MSEGO=12	;# segments owned by user
MSEGP=14	;# segments user is permitted to own.

Figure III.1 (continued)

block, MLNK, is a pointer to the name block for PPN [1,2]. In this name block is also the PPN which is associated with this block; in this case, [1,1] as well as a six -character password uniquely associated with the PPN. The name block contains a pointer, MAA, to an accounting block which gives information on the use of the system by users which are logged in under this PPN. This name block also contains a pointer, MUFD, which points to the header block associated with the user file directory for project programmer [1,1]. For every PPN in the system there is a unique name block in the Master File Directory which gives the PPN as well as the password and pointers to an accounting block and the UFD header block for that particular PPN. The first name block in the Master File Directory would be associated with PPN [1,1]. The second entry will be associated with PPN [1,2]. After that, the pointers to the next name block go to successive PPN's in an arbitrary order. The accounting block in the Master File Directory provides a convenient place for keeping track of information relative to any given PPN. In the accounting block are words which will give the total accumulated CPU time under that account (MCPU), the accumulated connect time for that account (MCON), the accumulated device time (MDVT), the accumulated system ticks (KCT), the number of segments owned by the user (MSEGO), and the maximum number of segments the PPN is permitted to own (MSEGP).

b. User File Directory

A User File Directory begins with a header block which is pointed to by the entry for that PPN in the Master File Directory. This header block points to the first of a linked list of name blocks for the

individual files associated with that PPN (see figure III.2). Every time a file is created, a name block is created in the User File Directory and linked into the list of active files in that directory. Also, an accounting block is created as well as the number of retrieval blocks necessary to allow the individual segments of disk related to that file to be located. As is shown in figure III.2 the first entry in each name block is a pointer to the name block associated with the next file. The last file in the list will have a link word of 0. The second, third, and fourth word of a file give the file name in RAD50. The first six characters is the name of the file. The next three characters is an extension. The next word gives the status of the file and protection associated with that file. The 6th word is an access count which tells the number of users currently logged in on that file. The last two words in the name block are pointers; first, to an accounting block for that file and second, to the first of a linked list of retrieval blocks for that file. The accounting block for a file will provide such information as the date of the last access, the number of segments in the file, and the date and time the file was created. The retrieval block contains in its first word a pointer to the next retrieval block in the chain or a 0 if this is the last retrieval block in the chain. The next words in a given block are the disk segments associated with that file. Since a file is a logically contiguous stream of information, the segments are logically contiguous beginning with the first segment (which is pointed to by the second word of the first retrieval block). The last word of the first retrieval block points to segment 7 while the first word of the 2nd retrieval block points to the 8th segment of the file. This continues to the end of the file. Note that while the

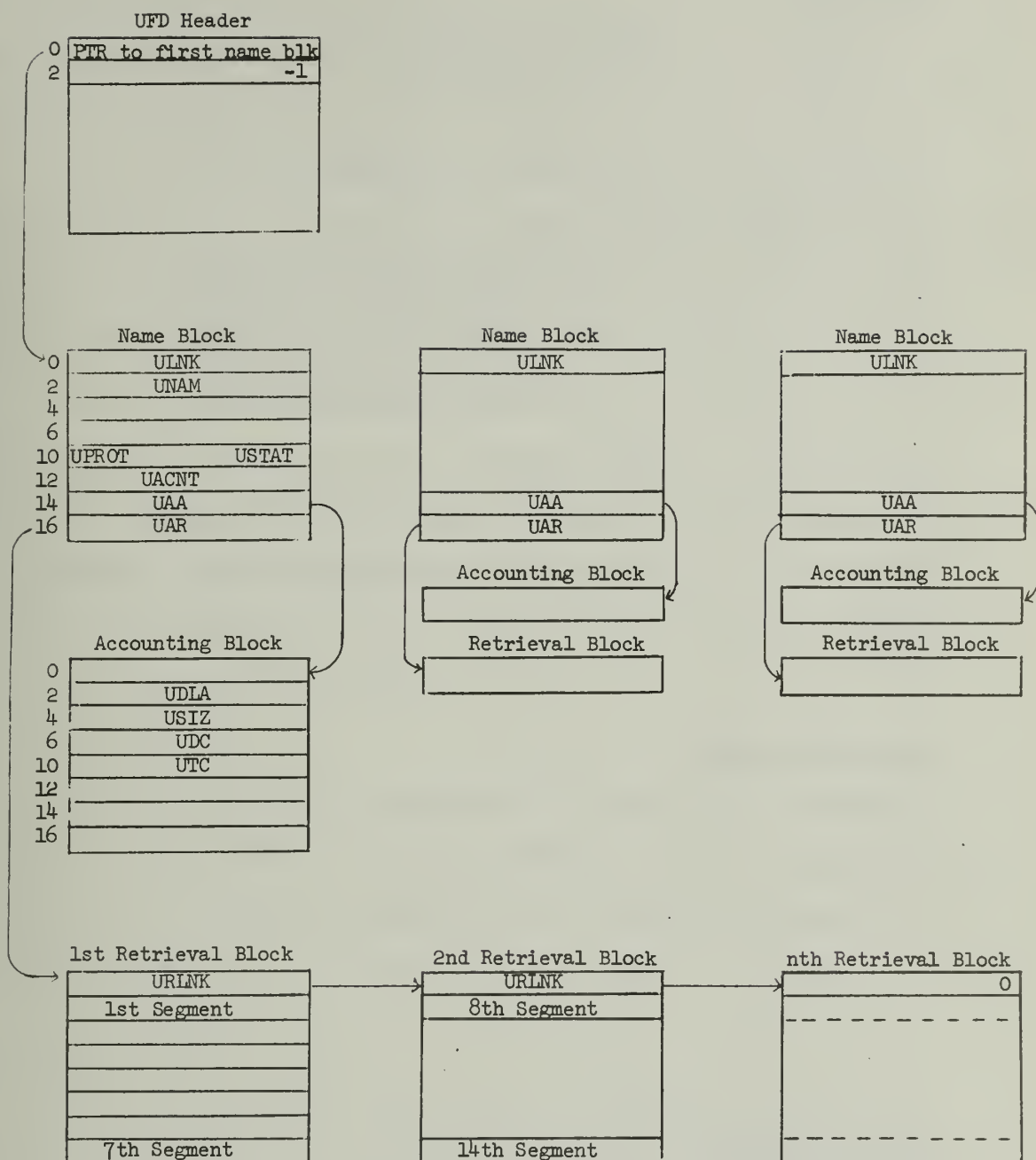


Figure III.2 User File Directory Structure

Name Block

ULNK=0 ;Link to next name block
 UNAM=2 ;3 words FILNAM.EXT
 USTAT ;Status Byte 200 if to be deleted
 100 if UFD
 UPROT=11 ;Protection Code
 40 - Write Protect from World
 20 - Read Protect from World
 10 - Write Protect from Group
 4 - Read Protect from Group
 2 - Write Protect from Self
 1 - Read Protect from Self
 UACT=12 ;Access Count
 UAA=14 ;Pointer to Accounting Block (Logical Address)
 UAR=16 ;Pointer to Retrieval Block (Logical Address)

Accounting Block

UDIA=2 ;Date of last access
 USIZ=4 ;# segments in file
 UDC=6 ;Date file was created
 UTC=10 ;Time file was created

Retrieval Block

URLNK=0 ;Link to next retrieval block

Figure III.2 (continued)

file is logically continuous in terms of these retrieval blocks, on disk it is any arbitrary set of segments of 256 words.

c. Directory Length

Both the Master File Directory and the User File Directory may consist of up to eight physical disk segments of 256 words each. The last eight words in each disk segment provides a list, in order, of the physical segment numbers of all segments in this directory.

The eight word header block of the Master File Directory is physically the first eight words of the first segment of the disk file area. Thus, the system finds the Master File Directory by looking in the header block in the first segment. Both the Master File Directory which is PPN [1,1] and the library which is PPN [1,2] reside somewhere in the first physical disk segment.

d. File Contents

FIP does not support any formatted files, nor does it maintain any type of parity information about the file itself. It does, however, support file name extensions which ETS' system software interprets to provide file types.

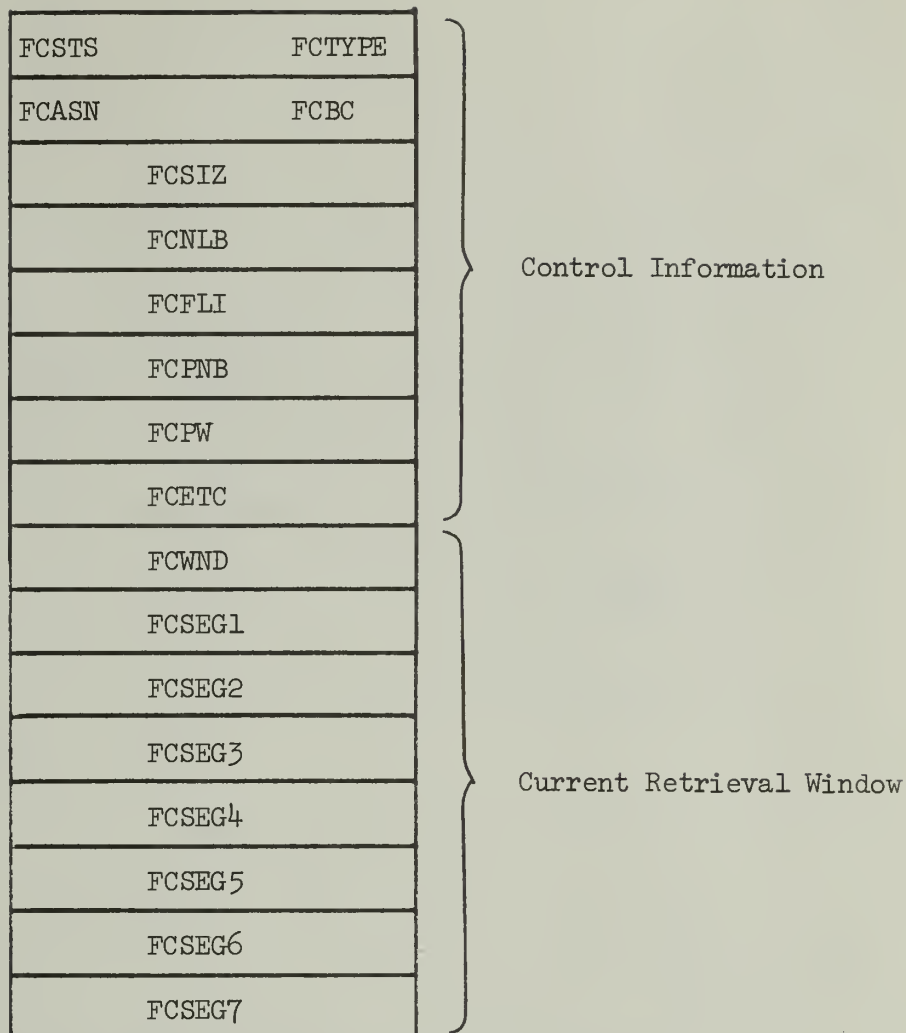
2. File Control Blocks

In any filing system one would like to minimize the overhead in doing device transfers. Since ETS makes many demands on the disk, a scheme to minimize the number of disk transfers is very important. This usually implies keeping much information in core--another resource in which ETS is severely limited.

The File Control Block (FCB) is a compromise which tries to minimize the number of disk transfers used to find pointers to blocks while minimizing the amount of core storage used for directories and retrieval lists.

Any time a user opens a file, a FCB (see figure III.3) is set up for that file which will enable the user to operate on the file without excessive disk transfers. Since the file has a linked list of retrieval blocks pointing to its segments, we keep a copy of one retrieval block (called a WINDOW) in the FCB. The variable FCFLI gives the logical segment number of the first block in the current window. Thus if the block we wish to read/write lies in the interval $[FCFLI, FCFLI+6]$, we simply subtract FCFLI from the block we want and use this as an index into the window to get immediately the disk segment # of the block we want. If the block we want does not lie in $[FCFLI, FCFLI+6]$ we are forced to "turn a window." Since we are normally operating in a forward sequential manner, FCWND, which contains the UFD address of the next window and FCPW which contains the disk address of the current window allow us to get a new window with normally only one additional disk access (see figure III.4). If we are not operating sequentially, and wish to move forward, we start from our present position and continue turning windows until we reach the window containing the block we want. If we want to move backwards, we must return to the name block which contains a pointer to the first retrieval block and search forward for the block we want.

The address of any file control block is kept in an I/O slot in the I/O block (see section II.B). Some file operations require that file be open (e.g., Protect, Rename) while some require that they not be open (e.g., Open, Create). Open files are referred to by their channel index (location of FCB pointer in IOB) while closed or non-existent files are referred to by name.

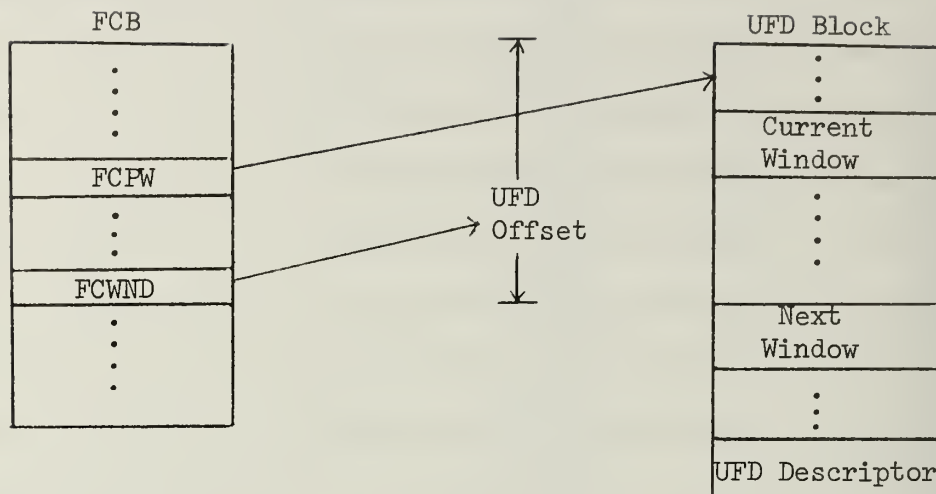


```

FCTYPE = 0      ;I/O handler index - disk = 0
FCSTS = 1      ;Status bits for file
                ;Bit 1 = 1 User is not owner of file
                ;Bit 2 = 1 User may not read file
                ;Bit 3 = 1 User may not write file
FCBC = 2      ;Block count for current transfer
                ;(He can only get 1 block at a time
                ;unless he does a direct access read/write)
FCASN = 3      ;Location of file name block in UFD (word offset)
FCSIZ = 4      ;# of segments in file
FCSIZ = 6      ;Logical block number of next block to read/write
FCFLI = 10     ;First logical block in window
FCPNB = 12     ;Disk physical segment containing name block
FCPW = 14     ;Disk physical segment containing current window
FCETC = 16     ;Miscellaneous word. Users doing a direct
                ;read/write put current memory address here
FCWND = 20     ;UFD address of next retrieval window
FCSEG1 = 22    ;Physical segment corresponding to FCFLI
FCSEGi = 36    ;Physical segment corresponding to FCFLI+i

```

Figure III.3 File Control Block



FCPW contains the physical segment number of the disk block which contains the current window. This block is a UFD block belonging to the User File Directory in which the file is listed.

FCWND is a "UFD Address" i.e., the low byte of FCWND is an offset in words from the beginning of a UFD block to the start of the particular retrieval window. The upper byte contains a number in $[0,7] * 2$ which is the particular UFD block containing the window. This is used as an index in the last 8 words (UFD descriptor) of any UFD block which contains the

segment # of the UFD block in question. If the new block is the same as the current block (the normal occurrence) no additional disk accesses are necessary. If not, one additional access is necessary to get to the block containing the new retrieval window.

Figure III.4 UFD Window Turn

B. File Processor

1. Basic Design

As is shown in figure III.5, FIP contains a major loop such that each function executed requires one pass around the loop. Since FIP can do only one thing at a time, all function requests are placed in a queue and FIP will cycle through them picking up a new request each time.

FIP consists of both core resident and disk resident code. Before FIP attempts to execute a new function, it checks to see if the function is in core. If it is not in core, FIP must read it into a special buffer (FIPBUF) before execution. FIP also has a second buffer (FIBUF) for use in storing data brought in from disk.

The FIP processor is self-sufficient in the sense that its only input is in the form of a File Request Queue Block (FIRQB) set up by the caller. It needs no device I/O other than disk, and uses no system resources other than CPU time and a few system buffers. It maintains its own stack which it switches to when running and switches back to the current stack when stopping (either for process completion or to wait on a disk transfer). With the exception of the EMT processor and RTI3, the system is not conscious of the presence of FIP. Figure III.6 shows the control data which FIP uses to define its own state.

Anytime FIP is running, it switches to a private stack. This is necessary to allow FIP to stop itself for disk transfers and to restart when they are complete, without worrying about the contents of its stack. Since the processing of each function is an independent task, when FIP completes a function, it throws away the contents of its stack; thus, it does not always worry about popping the stack as often as pushing. This is primarily

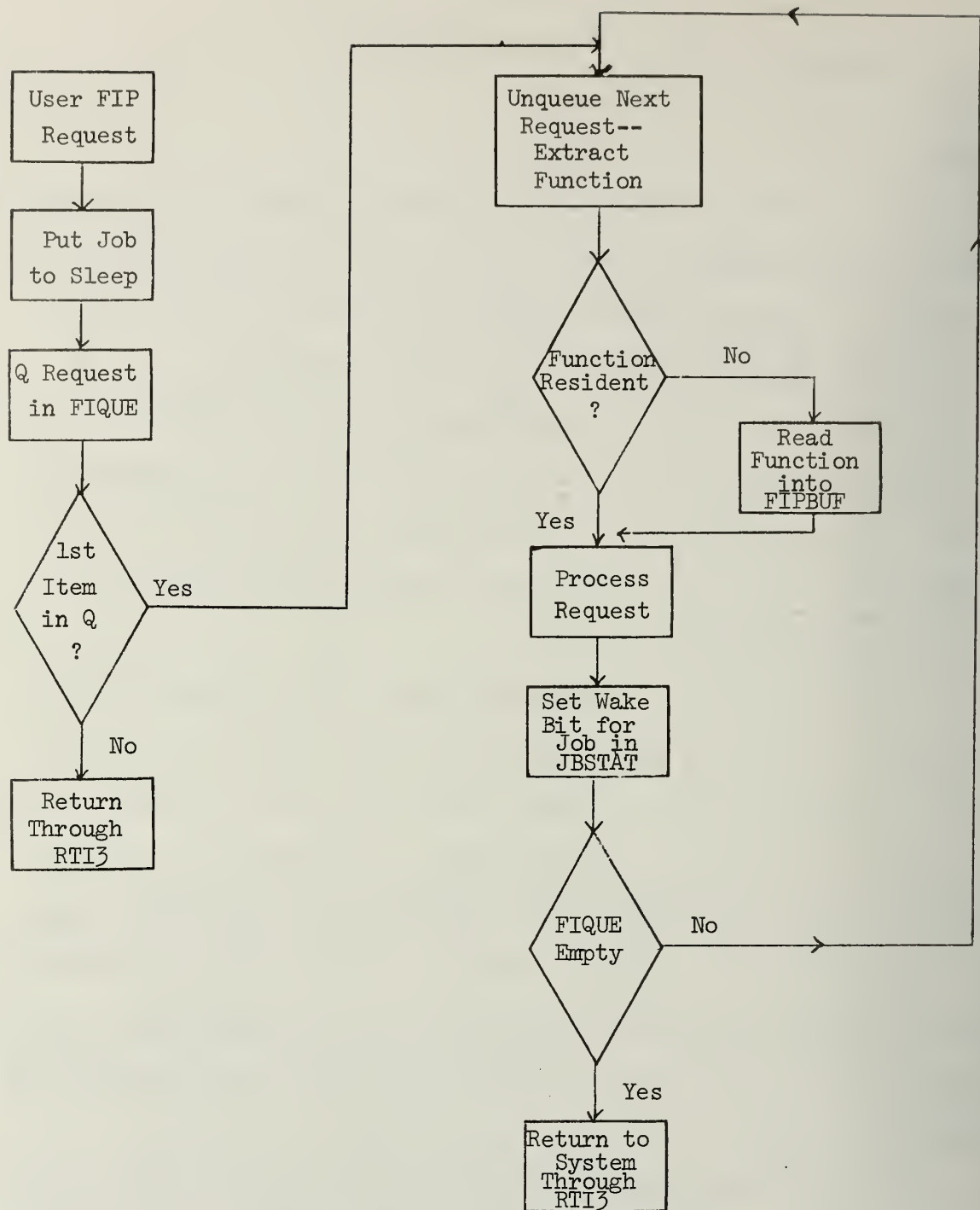


Figure III.5 File Processor Flow Diagram

FIQUE	-	Queue head for list of requests waiting for service.
FIBSTA	-	Buffer status of FIBUF. If the buffer is altered from what was brought in from disk, it must be written back to disk. = 0 unaltered ≠ 0 altered
FIUSER	-	Project-programmer number of job FIP is currently servicing.
FIJBDA	-	Address of job's first job data block (JDB).
FIPSEG	-	Non-resident code segment in FIPBUF (byte data).
FIBLOG	-	Logical segment index (*2) in FIBUF. A file directory (UFD or MFD) may consist of up to 8 logical segments.
FIBPHS	-	Physical disk segment in FIBUF.
SEGCNT	-	# of free (unallocated) segments in disk file area.
SATPTR	-	Cyclic pointer to disk bit map (SAT = Surface Allocation Table).
SATDEX	-	Segment number corresponding to SAT (SATPTR).
FIPSP	-	Current FIP stack pointer--valid when stack is system or user stack.
FIPR6	-	Current system stack pointer--valid when stack is FIP stack.
FIPAR	-	Disk parameter block for only transfers required by FIP.
DIRSIR	-	Routine to search a directory for a file name or UFD. Called by routines who need to ascertain the existence of a file or UFD.
GETUFD	-	Routine to read the start block of a UFD into core.
EXTOO	-	Routine to allocate a disk block and add it to a file by updating the retrieval blocks.
DELEET	-	Routine to delete a file and return its blocks to the system.
FQTOFC	-	Routine to pick up a FCB and IOB pointer given a FIRQB pointer.
GETNAM	-	Routine to get the name block of a file into core and to construct an absolute pointer to it.
UFDADD	-	Routine to convert an absolute address into a logical UFD address.
GETSEG	-	Routine to allocate a disk segment and mark it in bit map.
RETSEG	-	Routine to deallocate a disk segment.
READ	-	Routine to read a segment into FIBUF.
WRITE	-	Routine to write FIBUF back to disk if FIBSTA ≠ 0.
GETHOL	-	Routine to search for an empty 8-word block in a directory.
GETLNK	-	Routine to read segment given by UFD link into buffer and set up
GETLKS	-	absolute address.

Figure III.6 FIP Control Information and Utility Routines

used in error exits where FIP simply traps to the error handler. Note that interrupts may occur while on the FIP stack, but since FIP runs at priority 3, any interrupts will return control when their processing is complete.

2. Support Routines

As mentioned before, FIP consists of both resident and non-resident code. The criteria for determining whether a routine is to be resident or non-resident is the interdependence of routines. A routine may be non-resident only if it is not called by routines other than the non-resident code handler or other routines which reside in the same non-resident block.

The resident code generally consists of utility type routines--disk allocation routine, address mapping routines, segment reading routines, etc. Figure III.7 gives a list of many of the FIP utility routines and their functions.

C. File Requests

FIP uses a standard format for all file requests. A user who wishes service from FIP must allocate a system buffer (called a FIRQB) and supply in it the appropriate parameters (see figure III.7). He then executes an EMT (CALFIP=104050) to call the file processor. At the time of the call, R4 points absolutely to the FIRQB. On return, the buffer may or may not be returned depending on the function used.

FIP supports the following functions:

1. EXTFQ (=0) - Extend the file open on a channel given in FQFIL by an amount specified in FQSIZ.
2. WINFQ (=2) - Turn a window on the given file (FQFIL) to include the block pointed to by FCNLB.
3. DELFQ (=4) - Delete the open file given by FQFIL.
4. CLSFQ (=6) - Close an open file, the FCB is returned to the system.

5. OPNFQ (=10) - Open the file at FQNAML on the channel in FQFIL.
6. CREFQ (=12) - Create a file by name FQNAML of length FQSIZ-- default size is one block.
7. RENFQ (=14) - Rename the file at FQNAML to the name given in FQNAM2.
8. LINFQ (=16) - Log a user in. The PPN is given in FQPPN1, the password is given in FQNAML.
9. BYEFQ (=20) - Log a user out.
10. DIRFQ (=22) - Return directory information on the nth logical entry (specified by the word at FQSIZ) in the directory called in FQPPN1.
11. PROFQ (=24) - Set protection on file at FQFIL to that given in FQPROT.
12. PASFQ (=26) - Create a new user account. Current user must be logged in under account [1,1]. The new proj-prog is given in FQPPN1 and the password is given in FQNAML.
13. ERRFQ (=30) - Return an error message, specified by number in FQERNO, from the error file.
14. OPTFQ (=32) - Parse an instruction into its component parts.
15. LOAFQ (=34) - System loader--loads file given by FQFIL in user area with an offset given by FQSTRT. User must execute an EMT .LOCK before calling FIP.
16. ASSFQ (=36) - Assign a device--presently disabled.
17. DEAFQ (=40) - Deassign a device--presently disabled.
18. RSTFQ (=42) - Close all I/O channels except user teletype.
19. DAIFFQ (=44) - Deassign all devices--presently unnecessary since assign is not in use.
20. CSIFQ (=46) - Parse a file name in the form of DEV:FILE.EXT[N,M] and pack into appropriate FIRQB positions.
21. SWIFQ (=50) - Parse a string of switches and parameters and return a buffer of results. String is pointed to by FQPTR and is of length FQLEN.
22. FNDIL (=52) - Routine to determine whether or not a file/device exists and if so whether or not it is open and the channel on which it is open.

<u>Name</u>	<u>Value</u>	<u>Length (bytes)</u>	<u>Purpose</u>
FQQQ	0	2	Queue word for chaining requests.
FQJOB	2	1	Job # issuing request (#2).
FQFUN	3	1	Function requested.
FQFIL	4	1	Channel index.
FQERNO	4	2	Error or message # requested from file.
FQSIZ	5	1	Size of file to be created or number of segments to extend file.
FQPROT	5	1	New protection code.
FQPPN1	6	2	Proj-prog number of UFD desired.
FQNAM1	10	6	File name and extension in RAD50.
FQPPN2	16	2	Proj-prog number of second UFD.
FQNAM2	20	6	File name and extension for second file.
FQBLCK	22	2	Number of absolute loader blocks loaded.
FQCKSM	24	2	Number of checksum errors encountered.
FQSTRT	26	2	Offset from beginning of user area to "relative 0" of file to be loaded.
FQDEV	30	2	RAD50 device name.
FQDEVN	32	2	RAD50 device number.
FQLEN	34	2	Length of string to be parsed.
FQPTR	36	2	Absolute pointer to buffer containing source string to be parsed--string is in chained set of system buffers.

(Note that FQJOB and FQFUN must always be supplied--all other parameters need be supplied when actually needed.)

Figure III.7 FIRQB Structure

D. Errors and Protection

In order to help insure the integrity of disk files, FIP does extensive error and protection checking in file operations.

A user who wishes to modify a file must be write enabled on the file. ETS supports three levels of protection (see figure III.8). When a user attempts to open a file, the OPEN processor checks the protection and sets protection bits in the FCB accordingly. A file marked protect from self for both read and write can only be opened by its owner.

Anytime FIP finds an error, it will return an error code in the third word of the job data block (JDIOST). All EMT's below EMT 50 are FIP error calls. These must be used by FIP only, and they provide a means through which FIP can abort a function. Figure III.9 gives a list of the errors and their meanings.

<u>Code Bit</u>	<u>Meaning</u>
40	Write protect from world.
20	Read protect from world.
10	Write protect from group.
4	Read protect from group.
2	Write protect from self.
1	Read protect from self.

Note: Self implies other users with same PPN as owner. Group implies users with same 8-bit project #. World implies all others.

Figure III.8 File Protection Codes

<u>Error Name</u>	<u>Error Number</u>	<u>Meaning</u>
BADCMD	1	Invalid command.
BADDIR	2	Directory screwed up.
BADNAM	4	Badly formed file name.
NOTFND	6	File not found.
INUSE	10	File of same name is in use.
NOROOM	12	Directory is full or disk is full.
NOSUCH	14	File or UFD does not exist.
NOTCLS	16	Can't open already open channel.
NOTAVL	20	Device not available to user.
NOTOPN	22	Trying to operate on unopened file/device.
PRVIOL	24	Protection violation.
EOF	26	End of file on read/write.
ABORT	30	Operation aborted.
DATERR	32	Data error on device.
HNGDEV	34	Hung device for job.
NOLOAD	35	No load address specified on object file.
CKSMER	36	Checksum error in loader.
OUTRNG	37	User area not large enough.

Figure III.9 FIP Error Calls

IV. I/O SUBSYSTEM

A. Basic Principles

1. Data Representation

ETS supports two types of data representation:

Seven-bit ASCII is used for the internal representation of character or string data. With the exception of cards, it is also the representation of such data on off-line storage mediums. Cards use the EBCDIC character set and are translated to ASCII by the card reader handler. Character data is handled as a stream of lines or blocks.

Data which is not specifically ASCII is called binary. Binary data is full eight-bit character representation and has no preassigned meaning for any character or sequence. Consequently, when handling binary data, we depend on the device to inform us when we have reached the end of the text.

2. Data Transfers

There are two modes of data transfer:

Line-oriented transfers may be used when the data is in ASCII form. This type of transfer is used exclusively from the teletype when a program is inputting data. In a line-oriented read, the end of the transfer is determined by either the occurrence of a terminal character such as a carriage return, or the fulfillment of the maximum requested transfer length.

Block-oriented transfers are the normal mode and associate with each transfer a 'transfer length' which must be fulfilled in order to terminate the operation. When transferring to/from disk, block lengths are fixed by the system at 256 words. For other devices, the block length may be set by the user.

3. Device Independence

In order to provide a reasonable degree of user convenience, ETS attempts to provide a measure of device independence. Device I/O requires the user to perform three basic steps. First, he must request the use of a device or file; he does this by calling the file processor to 'open' a file or device for him. Second, he must request the actual read/writes. Third, he returns the file/device to the system by again calling the file processor to close it for him. ETS uses three basic mechanisms to provide this device independence:

When a file or device is opened, it is associated with the user job by assigning it to a 'channel' or 'slot'. Each job may have up to eight active channels. Once a device is opened on a channel, the user may refer to it via the channel number and he needs only a vague idea of what type of device he is connected with.

When performing I/O, a user must provide the system with a transfer control block (XRB) which contains the parameters to be used in the transfer of data. This control block contains only a channel number, a pointer to a buffer and transfer count, and possibly a few parameters. In general, the user may use the same XRB to control more than one device.

Each of the ETS I/O handlers consists of two co-routines. The first is the device service routine which is responsible for the

communication between the user and the system. The device service routines (in the case of input) will take characters from system buffers and transfer them to the user. If the buffers become empty, the service routine will initiate the device interrupt service in order to refill the buffers. There is a specific interrupt handler for each device which is capable of catering to the special needs of that device. The device service routine can then worry about taking the data as returned by the interrupt handler and converting it to the standard system format.

B. Data Structures and Control Routines

1. I/O Control Structure

The basic control structure for an I/O device is the device data block for everything except disk, and the file control block for the disk. When a user opens a file/device, a pointer to the DDB/FCB is placed in one of his channels. In the case of devices, the user's job number is also placed in the DDB so that his job control blocks may be located if the DDB address is known. The file control block is described in section III.A.2. The device data block has an organization which is unique to the particular device. All DDB's have common headers which provide type and status codes and the job number of the owning job. Second, DDB's contain the pointers which control the monitor chain buffers for that device. Third, DDB's contain any special control information required for the proper operation of a device. (figure IV.1 is the DDB for the line printer.)

2. Character Handling--Monitor Chain Buffers

With the exception of the disk, all devices operate on a character by character basis, i.e., the interrupt handlers get an interrupt

LPTSTS	LPHAND
LPJBNO	
LPTIME	
LPVERT	LPHORZ
LPTFP	
LPTFC	
LPTEP	
LPTEC	
LPTBC	
LPTAB	
LPCNT	

LPHAND	= 0	;Handler index
LPTSTS	= 1	;Status
LPJBNO	= 2	;Owner job number
LPTIME	= 4	;Time device assigned
LPHORZ	= 6	;Horizontal line position
LPVERT	= 7	;Vertical page position
LPTFP	=10	;Chain buffer fill pointer
LPTFC	=12	;Chain buffer fill count
LPTEP	=14	;Chain buffer empty pointer
LPTEC	=16	;Chain buffer empty count
LPTBC	=20	;Chain buffer available buffer count
LPCNT	=36	;Init count

Figure IV.1 Line Printer Device Data Block

for each character transferred. In order to simplify the reading and writing from the user programs, we want to transfer blocks or strings of characters to the device service routines. This allows us to swap the user programs out for the duration of the transfer. We implement this buffering through the use of the monitor chain buffers.

Each monitor chain buffer requires five words of control in the DDB. The first four words simply implement a first-in first-out queue while the fifth gives the maximum size of the queue. The queue is implemented as a set of buffers which are chained together through their first words. As one buffer is filled, another is allocated and added into the chain. When a buffer is emptied, it is returned to the system. The chain buffer discipline is maintained by two routines called FETCH and STORE. Since either the filling or the emptying (depending on the type of device) of the buffers is under interrupt control and cannot therefore be anticipated, it is necessary to provide interlocks between FETCH and STORE to prevent misuse of the pointers. The simplest interlock is to have no overlap at all between the routines. This is done in ETS by running both routines at processor priority seven.

3. Transfer Control Blocks

User programs control the transfer of data by means of a transfer control block (XRB). The XRB tells the system the address and length of the user's buffers as well as the channel to which the I/O is to take place (see figure II.5). When a user is reading from a device, he must give the system a buffer address as well as a maximum length for the transfer. In the case of a block-oriented transfer (e.g., card reader), the system will continue reading until it fills the user buffer unless there is an error

or end-of-file during the operation. In the case of a line-oriented read, the system will transfer characters until an end-of-line is encountered or the buffer is full, whichever occurs first. When writing to a device, the system will transfer the exact amount given by the user in the byte count field of the XRB unless an error occurs. In the case of operations to/from disk, all transfers are exactly 256 words.

In actually executing the transfer, the system may not be able to fulfill the entire request at one time; consequently, it will transfer the maximum amount possible and put the job to sleep while the system performs physical I/O to catch up with the job. In order to pick up from the point at which this logical transfer leaves off, the system backs up the user's program counter before putting him to sleep so that when the user is awakened, he will immediately re-execute the transfer. The system modifies the XRB pointers during the transfer so that at any time, they show exactly how much information remains to be transferred.

It should be noted that the system may be able to complete the logical transfer of information between the user and the system buffers before it can complete the physical transfer. Even though the actual physical transfer may not be complete, the user program is allowed to continue running. Should it then execute another I/O transfer, it is possible that that operation will complete before the first (e.g., write to TTY followed by a read from the TTY in which a message has already been typed). This effect allows full duplex communication between a user program and its devices. This can in turn cause such things as the sequence in which typing appears on a teletype to disagree with the order in which the program performed reads and writes; however, it will not affect the execution of the user program.

Since the execution of a read/write may take a large amount of real time but virtually no CPU time, the system may decide to swap a user out while it catches up with some physical I/O. When it swaps the user back to core, it may relocate him, consequently, all XRB pointers are user-job relative.

4. Hardware Priority Arbitration

In order to insure that devices may interrupt and be serviced in an amount of time commensurate with their need, they are assigned to different levels of hardware interrupt. Since the card reader has a fixed amount of time in which it must be serviced, it is given the highest level in the system. Similarly, the teletypes, line printer, etc. are not sensitive to time and are given lower levels. Figure IV.2 gives the hardware priority of each of the devices on the system. Also shown is the processor priority which is used by the interrupt service for each of the devices.

C. I/O Characteristics of Individual Devices

1. Disk

I/O to/from the disk is in the form of files. As described in chapter III, files are sets of logically contiguous blocks with 256 words/block. All transfers involving the disk are of length 256 words. Successive transfers to/from a particular file advance a system pointer to the next block in the file thus giving the effect of reading a file sequentially. This is what a user normally wishes to do and is thus the default action; however, users may desire to transfer specific blocks of a file (e.g., each block contains a different segment of non-resident code).

<u>Device</u>	<u>Hardware Interrupt Level</u>	<u>Interrupt Service Priority</u>
Card Reader	6	6 (drops to 5 after character in tumble-table)
Clock	6	5
Disk	5	5
Line Printer	4	4
Papertape Reader	4	4
Papertape Punch	4	4
Teletype	4	4

Figure IV.2 Device Priority Assignments

The system provides an EMT routine called .FIND which may be used to position a file at any desired position. By executing a .FIND followed by a read/write the user may transfer any part of the file he desires. Should a user attempt to transfer to a block which does not exist, the system will generate an end-of-file error. The user may then call a system function to extend the file and try again.

2. Papertape Reader/Punch

The papertape reader/punch is the simplest of the I/O devices. All transfers are block oriented and will complete the transfer unless the reader or punch runs out of tape. No validating of the data is done by the system.

3. Card Reader

The card reader is possibly the most complicated of the devices on the system, both from the standpoint of the data manipulation it does, as well as the control of the card reader by the system.

The card reader accepts parameters from the user XRB which tell it the type of data to be read. The user may read EBCDIC as punched on an IBM 029 keypunch in which case the driver will convert it to the equivalent ASCII representations, or he may read in binary mode in which case the card reader will transfer directly the top eight rows of a card column as one byte. This mode is useful for reading the absolute loader format decks produced by the PAL-11 assembler on the IBM 360. When reading EBCDIC cards, the user may also specify card margins in order to read only those columns in which he is interested.

The card reader interrupt driver must be able to respond to interrupts at the rate of one interrupt every 800 microseconds. In order

to allow a reasonable safety factor in the time requirements, characters are placed in a 16 word tumble-table as they are received. Until the character is actually in the table, interrupts are not allowed, which takes approximately 100u secs after which the reader is able to accept another interrupt. The characters are removed from the tumble-table in order, the desired translation is performed and the translated character is stored in the monitor chain buffers. By using this tumble-table, we need only insure that the average processing time for a character is less than 800u secs rather than the maximum time (the worst case max time is approximately 800u secs). The interrupt driver also attempts to save time when in EBCDIC mode by doing blank suppression. When a blank is received, it is simply counted. When the next non-blank character is received, the current number of blanks is stored (as a negative number to avoid confusion with regular characters which are always positive) followed by the character. The card reader to user interface will then expand the blank counts back into actual characters. This technique also makes trailing blank suppression trivial since all training blanks will be contained in a single byte which may be ignored.

4. Line Printer

The line printer looks somewhat like the papertape punch except that it does some character interpretation in order to provide formatting ability. The driver interprets tabs, line feeds, carriage returns, and form feeds. For the user, output to the line printer requires only an XRB with no parameters. The only consideration to worry about in the line printer interrupt driver is the speed with which the line printer may request interrupts. The line printer buffers internally up to twenty-four characters

and consequently may request them as fast as the system is able to supply them; this means that the interrupt handler must run at a high enough priority to block subsequent interrupts until it is ready for them. It should be noted that unlike the card reader, the line printer will wait indefinitely for the next character.

5. Teletype

The only complicated part of the teletype I/O drivers comes from the fact that the teletype functions as both an I/O device and as a job control device. Consequently, characters received from the keyboard must be checked to determine their meaning. In general, characters are interpreted as simple line input which looks like any other I/O device. The system does check each character input against a list of 'control characters' to see if it requires special action. This is accomplished by using a system routine called TTSIVE which will check if a character is between a given set of limits; if not, it will compare it against a list of special characters and call a corresponding service routine if a match is found.

The teletype keyboard and printer operate in full duplex mode which means that a user may be typing in one message on the keyboard while another message is printing on the printer. Since the printer will print characters as they are received by its service routine and placed in the monitor chain buffers, it is possible to have the echoes from the keyboard interspersed with the output from a user program.

D. Error Recovery

ETS does not attempt sophisticated error recovery. In general, ETS will terminate the current operation as quickly and neatly as possible and return an error code to the user program.

In the case of the disk, ETS will re-attempt the transfer up to five times before returning the error. In the case of the card reader, the current transfer will be stopped and the eject command will be issued for the current card. This works well for source data since the user program can simply discard everything since the last line ended. The user may then reinsert the last card and re-execute the transfer. Things are somewhat messy when doing binary transfers; the best solution is to simply start all over.

When an error occurs, ETS only returns the code to the appropriate user; it is the program's responsibility to type out the message if that is the desired action.

V. COMMAND STRING INTERPRETER

The Command String Interpreter is probably the simplest part of the ETS monitor. The monitor assumes that a user always has a job which is capable of running. When he first enters a command, he has not chosen what he wants to do, so the system connects him with what is in effect a default job which is the Command String Interpreter.

The Command String Interpreter simply provides the user with a minimum set of operations to allow him to connect himself with the system and to initiate one of the system programs. These operations are:

Login. When a user first sits down at a console, he must identify himself to the system. This identification will establish whether or not he is a valid user of the system, and if so, allow the system to locate the appropriate user file directory. Until he successfully logs in, the system will not honor any other requests a user may make.

Time and Date. The system provides commands which return the present time and date to the user.

Core. When the system first establishes the presence of a new user, it will allocate a default region of 1024_{10} words of core. Since this is insufficient for most programs, the Command String Interpreter provides a command through which a user may request a new region size. This command is useful for either increasing or decreasing a user's region size.

Run. Since all useful work by a user is accomplished using one of the system programs, the RUN command is used to initiate the execution of

one of the system packages. This command will first of all check that the requested job is one of the system library programs (users may not run their own programs directly). It will then close any I/O channels which the user may have left open from previous operations and call the system loader to load the specified program. If the program is loaded successfully, control will be transferred directly from the Command String Interpreter to the specified program.

Logout. When a user has completed his work session, he tells the system of his desire to quit through the logout command. The system will then tidy up after the user by making sure that all resources have been returned and terminate the job.

Other operations which are not actually part of the Command String Interpreter but are used to control the execution of a job are the teletype characters which receive special processing by the teletype interrupt handlers:

↑C (Called Control C and is typed by holding down the CNTL key and striking C). This command causes the system to set the abort flag on the user's job. The next time the scheduler selects this job, it will refresh the stack and return control to the Command String Interpreter.

↑QUIT (CNTL Q). This command is the normal means of terminating the execution of a system job. The system will set a flag for the user program. When the program recognizes the flag, it may tidy up after itself and then return control to the system. This is much better than ↑C since the program may close files and write out information which the system would normally be unable to recover.

↑V (CNTL V). This works in a manner similar to ↑QUIT except that it does not request that the program terminate. It is normally used to request that a program return to its own command mode as the user has made some type of error in his input (e.g., the program is in an infinite loop because of a user input).

↑O (CNTL O). This is a system command which will cause the termination of printing of programmed output on the teletype. If a user performs a function which causes an unexpectedly large volume of output to be printed, he may ignore it by using ↑O. The effect of this command is terminated by a TTY reset or the typing of a carriage return.

↑B (CNTL B). This is a useful device for preventing the echoing of code words. The command operates a toggle switch (each occurrence changes the state of the switch). When the switch is on, nothing that is typed is echoed on the TTY printer.

The Command String Interpreter is simply an always resident reentrant job. It is composed of a central loop (NEWGUY) which will accept a string from the teletype and use the first two characters of the string as a command. Based on those characters, it will dispatch to the appropriate routine to process the command. The major difference between the CSI and a normal program is the way it does I/O. The Command String Interpreter performs its I/O at a much lower level than the normal case. This is necessary to allow it to pick up only a part of a line at a time from the chain buffers. In general, each part of the Command String Interpreter picks up only those characters which it needs to do its specific processing.

The Command String Interpreter also provides a central error routine which may be called from any program. This routine will get the

specified message from the system message file and type it out on the teletype. Since the messages are stored on disk, this provides a convenient means having a number of messages available without using excessive storage to keep them.

Figure V.1 gives the command format and a list of accepted commands.

<u>Command</u>	<u>Interpretation</u>
HELlo 2 7 PASWRD	Log onto system under user code [2,7] which has code password PASWRD
SIze 3	Request the allocation of 3K (3072_{10}) words of core
TIme	Request for the present time
DAtE	Request for today's date
RUn EDETS	Request to load and execute the system editor
MEssage 23	Asks the system to give the meaning of message #23
BYe	Log out, user done

(lower case letters need not be typed)

Figure V.1 System Commands

VI. PERFORMANCE AND IMPROVEMENTS

A. Performance

At the time of this writing, ETS has been operational for about a month; however, it has not been in heavy use as yet because it was felt that the switch from batch to timesharing in the middle of a semester might delay students too much. Also at this time, additional core memory has not been installed on our system. The addition of this core will have a major effect on the throughput of the system. As a result of these two things, we have not had time to collect extensive statistics on the performance of the system under normal loads for extended periods of time.

Even without extended data collection, we can learn much about the system performance from a few simple measurements.

1. Execution Time for Selected Segments

We can calculate the execution time of some of the heavily used sections of code and project the CPU time that will be spent executing those routines. Figure VI.1 gives the execution time for some common system routines. The execution time for device interrupt services is very important to the system since it must be handled in real time. In particular, the card reader and line printer can absorb a large amount of the available CPU time while running. The teletype interrupt service also requires a considerable amount of processing although the character

<u>Routine</u>	<u>Time (μsecs)</u>
Buffer Allocation	
Get 16 word buffer and clear it	275
Get 16 word buffer, clear first word	75
Buffer Deallocation	
Return buffer	100
FETCH Character from Monitor Chain Buffers	
Character in buffer and more than one block chain	42
Character in buffer and one block in chain	51
Character is last one in a block	145
STORE Character in Monitor Chain Buffers	
No buffers exist in chain	186
Buffer exists and is not full	81
Current buffer is full	241
Turn off all interrupts	23
Turn interrupts back on	18
Register Save	
Processor priority ≤ 3	77
Processor priority ≥ 4	30
Register Restore	
Processor priority ≤ 3	133
Processor priority ≥ 4	32

Figure VI.1 Execution Time for Some System Routines

transmission rate is low. We consider the effect of these three devices on the system:

a. Card Reader

We assume an average user who is reading a source language deck into the system. We assume that the cards have an average of about 40 characters. We count single blanks as characters and spans of blanks as single characters. We can then interpret the card as containing 46 'characters' and 34 blanks which have been compressed. Reading cards require two system routines: the card reader interrupt handler and the card reader user service.

The interrupt handler requires ~ 435 $\mu\text{secs}/\text{char}$ on the 46 'characters' and ~ 331 $\mu\text{secs}/\text{char}$ on the 34 blanks. This is an average of 391 $\mu\text{secs}/\text{char}$ for the 80 column card. (Note: This time includes 153 $\mu\text{secs}/\text{char}$ to translate the card code to ASCII. By using a 128 word table and the translate facilities of the card reader, this can be cut to 4.9 μsecs .) For a 600 cpm card reader, this operation requires about 31.4% of the available CPU time.

The card reader user service requires about 160 $\mu\text{secs}/\text{char}$ on the non-blanks and about 100 μsecs each for the blanks which are members of a span of blanks. This is about 129 $\mu\text{sec}/\text{char}$ for the 80 column card. This is about 10.3% of the available CPU. If trailing blanks are suppressed, this is closer to 8%.

Since much of the processing is simpler when copying binary cards, the total CPU commitment is closer to 25%. However, the reading of binary decks is almost negligible when compared with the number of source decks read.

b. Line Printer

We can assume that the user is getting an assembly listing with an average of 100 char/line. For lines of this length, the line printer can print about 256 lines/minute or 420 char/second. The line printer interrupt service requires about 198 μ secs/char to process the outgoing characters--which is about 8.5% of the CPU.

The line printer user interface requires about 160 μ secs/char to get the characters from the user and store them in the monitor chain buffers. At 420 char/sec, this is about 6.9% of the CPU.

Of all devices on the system, the card reader and line printer make the greatest demands on the system. However, when operating in timeshare mode (instead of batch) the amount of time that either is in use is small ($\sim 5\%$). Even with a batch partition, it would seem unreasonable to expect the actual time that the card reader is in use would go above 20%. However, at this level, the savings from using hardware translate facilities for the characters become appreciable.

c. Teletype

Because of the many interpretations a character from the teletype may have, they require more processing/character than any other device. From a low of ~ 400 μ secs for a character which requires no special processing they may require up to ~ 2000 μ secs for a control character which requires extensive processing and multiple echoes.

Even though they may require much processing per character, the CPU time necessary to support teletypes is small. We may assume 10 users who are all using their teletype at the maximum rate (a highly

unlikely circumstance). Since no extended sequence of control characters is meaningful, we assume 500 $\mu\text{sec}/\text{char}$ processing. The maximum transfer rate for a teletype is 10 char/sec. Thus, 10 users need not more than 5% of the CPU. Since 10% teletype usage is probably a better estimate, we really expect <1% of the CPU to be used for teletype I/O support.

d. Clock

The clock provides a continuous fixed overhead for the system. It interrupts every 16 msec and requires the execution of three instructions on 5 of 6 interrupts. Every 100 msec it requires about thirty instructions. This is a total of less than 300 $\mu\text{sec}/100 \text{ msec}$ or .3% CPU.

2. Response Time

The most important statistic about a timesharing system is response time. For a given input, how long must a user expect to wait before the system is ready to accept his next input. Unfortunately, this is the most difficult measurement to make. Response time is sensitive to many different variables including type of input, current job mix, probability of requesting an unavailable device, and current configuration of the hardware. For ETS, the last variable is currently the controlling factor in the list. With a user area of only 5K words and jobs that require the entire region, ETS is forced to sit idle much of the time while waiting for 5K jobs to swap in and out. It takes ~100 msec for a 5K disk transfer to take place, thus a 5K job requires 200 msec of disk delay in order to execute for one timeslice (100 msec). With a 12K user area, it becomes feasible to execute during the swap time of large jobs since we can expect to have two other jobs resident.

In a worst case experiment, we have run nine infinite loops each as a 5K job while attempting to run an editor or I/O bound job from the tenth terminal. The response for the execution of a simple edit command is of the order of three seconds. This is directly predictable from the amount of time it takes for the system to swap and run each of the other continuously executing jobs. With additional core, this should decline to 1-2 seconds.

In another worst case experiment, we executed continuous edit commands from all teletypes. Since the editor is a 2K job, response time was order 1-2 seconds depending on the amount of processing generated by a given command. The editor does a great deal of disk I/O when some of the more powerful scan functions are invoked; since a job must be locked into core during these operations, continuous disk operation by a job can block the CPU for extended periods. This condition is now avoided by causing such jobs to return control to the system for rescheduling at other than timeslice intervals (e.g., between I/O operations). When there are sufficient jobs resident in core, the CPU will be able to switch jobs while waiting for disk operations, thus alleviating this problem.

B. Tuning the System

While ETS is a correctly functioning system, it is not certain that it is making the best possible use of its resources. In particular, when it takes 200 msec to swap a large job to/from disk, it seems expensive to allow it to execute for only 100 msec. However, should we give each job a 200-300 msec timeslice, the last in a sequence of jobs awaiting execution may have a response time of five seconds or more. There are a number of control points which may be modified in order to maximize the throughput to response time tradeoff.

1. Varied Timeslices

Currently, ETS uses a fixed 100 msec timeslice. A first approximation involves running experiments which vary the size of the timeslice. A better solution involves a timeslice which is dependent on the size of the job, i.e., big jobs get more time. However, in order to avoid favoring big jobs, the scheduling algorithm must be modified to select small jobs when possible.

2. Disk Optimization

Disk segments are not allocated with any optimization in mind. For files which are read by a loader or other program which reads them at a more or less fixed rate, proper allocation of segments can reduce latency. Currently, transfers are 256 words in length which take 4 msec, since latency is 16 msec, file accesses are nominally 20% efficient. Of all the possible changes to ETS this is probably the most difficult and least feasible. However, considerable optimization is possible on disk accesses if the system were to select the nearest of a set of requests instead of the request which arrived first. This effect will not be beneficial until there is enough core to have a good probability of pending disk requests for multiple jobs. If there are three jobs with disk requests pending, it is likely that latency can be cut in half.

3. Order of Major System Functions

The system controls the order of the processing of its major tasks by associating a bit in the variable L3QUE with each. By changing the bit assignments, it is possible to change the processing order of system functions. It is very likely that the functions are now in the correct order although it would be an interesting experiment to change them.

4. Buffer Allocation

Each I/O device has a maximum number of buffers which it is allowed to allocate. The more buffers a device is allowed to use, the longer it can be kept busy without system intervention. The best scheme involves giving as many as possible to each device; however, the buffer pool is very finite and it is imperative that the system not run out of buffers. Consequently, throughput can be improved considerably by finding the maximum number of buffers which can safely be given a device.

5. Scheduling

The system currently schedules jobs in a round robin fashion. If priority is given to jobs which require much interaction (e.g., editor), larger timeslices can then be given to other jobs which need large amounts of CPU. The effect of this should be to increase the throughput while simultaneously improving the response time for most jobs. Since facilities exist for the naming of jobs within the system, such algorithms should not be too difficult to implement and should provide a handsome reward in terms of throughput.

C. Improvements to the System

ETS has reached a major landmark in its existence in that it is running and seems reliable. With a basic system in hand, it is reasonable to investigate improvements to the system. In addition to those improvements necessary to tune the system, there are some additional features which would greatly enhance its utility and power.

1. Dynamic Swap Area Allocation

ETS currently reserves a swap area on disk for each job which is equal in size to the largest amount of core which a job may ever request.

Since the average job size is probably less than half that, a large amount of disk space is being wasted. By establishing dynamic swap area allocation, this space can be recovered for the system. Implementing such a scheme basically entails incorporating a table with each job which contains a list of that job's core segments. The swap routines must then use this table when searching for jobs on disk rather than using the job number and the compile time disk allocation.

2. Buffer Accounting

When buffers are allocated to jobs, ETS does not keep a record of them in order to recover them in case a job is unexpectedly aborted. A form of accounting is needed in which the system keeps track of buffers which may be lost. Note that the system need not keep track of the majority of buffer allocations since they will go to I/O drivers which cannot be aborted. There is space in the last half of the I/O block to keep track of allocated buffers. This should be a fairly simple modification.

3. HELP

Since ETS is designed as an educational system, it should probably be able to help the student who is totally lost or uninitiated. In general when a key is typed and the system cannot understand the student input, it should default to an assistance package which will explain things to the student. With the exception of differentiating between the student who knows what he is doing but has made an error and the student who needs help, such a package is very simple to implement.

4. Sharable Procedures

Many programs which are written to run under ETS need similar subroutines and functions. Most of these could be implemented as FIP

processes, however, the overhead on such functions which were called repeatedly could be exorbitant. Alternatively, many routines could be made system resident and linked to the user via the TRAP instruction. This would allow a position independent call and would allow system routines to access one another. Routines which were not used often could be stored on disk and swapped into and executed from a buffer which is provided by the individual user. Such a mechanism could also be used to implement non-resident code segments for users who do not want to provide the controller or wish to swap their whole area.

5. Reentrant Editor

Additional improvements in response time for functions such as the editor can be achieved by incorporating them as resident reentrant parts of ETS. By simply coding the package as a part of the command string interpreter, routines may always remain resident and may use the user swap area as a data area. This method would make it simple to implement an editor which used only 1K of user area and would then be very fast to swap and easy to fit in core.

These and many other improvements to ETS are possible and would make excellent projects for students who are learning assembly language programming and operating systems. Most are not so large as to require more than one semester or so complicated as to be restricted to only the superior students.

BIBLIOGRAPHIC DATA EET	1. Report No. UIUCDCS-R-72-520	2.	3. Recipient's Accession No.
	Title and Subtitle The Design and Implementation of an Educational Timesharing System for the PDP-11		5. Report Date June, 1972
Author(s) Donald W. Oxley		6.	8. Performing Organization Rept. No.
Performing Organization Name and Address Department of Computer Science University of Illinois Urbana, Illinois 61801		10. Project/Task/Work Unit No.	11. Contract/Grant No. GJ-812
Sponsoring Organization Name and Address National Science Foundation Washington, D. C.		13. Type of Report & Period Covered Research	14.
Supplementary Notes			
Abstracts ETS is a mini-computer timesharing system designed for use in introductory assembly language programming courses. Its purpose is to provide powerful facilities for beginning students to use in constructing and debugging assembly language programs. It is written in the PAL-11 assembly language for use on the PDP-11 computer.			
Key Words and Document Analysis. 17a. Descriptors Timesharing System, Operating Systems, Mini-computers, Multiprogramming			
7 Identifiers/Open-Ended Terms			
7 COSATI Field/Group			
8 Availability Statement Unlimited Release		19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 80
		20. Security Class (This Page) UNCLASSIFIED	22. Price

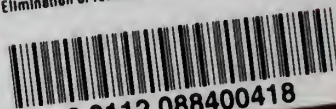
JUL 26 1972



UNIVERSITY OF ILLINOIS-URBANA

510.84 IL6R no. C002 no. 517-522(1972)

Elimination of rotational latency by dyn



3 0112 088400418